L4: Entry 2 of 2 File: USPT Jul 2, 1996

DOCUMENT-IDENTIFIER: US 5533148 A
TITLE: Method for restructuring physical design images into hierarchical data models

Brief Summary Text (26):
The method includes the steps of providing an image, identifying leaf areas in the image, classifying the leaf areas as being of one or more types according to a characteristic of the leaf areas, grouping adjacent leaf areas of equivalent type into one or more area arrays, recognizing equivalent area arrays and forming a set of one or more unique area arrays, partitioning each unique area array into one or more partition areas, recognizing equivalent partition areas and forming a set of one or more unique partition areas, and generating area specifications for the entire image area, for each unique partition area and for each unique leaf area.

Brief Summary Text (27):
In one embodiment of the invention, the step of grouping adjacent leaf areas into area arrays includes the steps of scanning the image in a first direction, grouping leaf areas of equivalent type that are adjacent in the first direction to form one dimensional area arrays, scanning the image in a second direction, and grouping equivalent one dimensional area arrays that are adjacent in the second direction to form the area arrays.

Drawing Description Text (7):
FIG. 5(a)-5(d) illustrates a second sample physical design image used in describing the steps of identifying leaf areas and grouping the leaf areas in the method of the invention.

Drawing Description Text (10):
FIG. 7 graphically illustrates the steps of grouping leaf areas into area arrays and identifying the unique area arrays.

Detailed Description Text (3):
The "area specifications" of the model describe the characteristics of areas on the image 100 which are important to an application program seeking to use the areas, e.g. for placement or wiring. The areas are hierarchically organized in an area tree 300 which is similar to the prior art quad area tree, previously described, in that there is a one-to-one correspondence between nodes in the tree and defined areas. The node 302 at the highest level designates the entire image area and the nodes (e.g. 322) at the lowest level (called "leaf nodes" because they are at the ends of the branches in the area tree) designate the smallest areas (referred to as "leaf areas").

Detailed Description Text (14):
In the hierarchical area specification model, however, all of this information is stored in a single area specification indicated by oval 214. Since there are only two fundamental area types at the lowest level, there are only two area specifications, 214 and 224 (referred to as "leaf area specifications"), at the lowest level of the area specification model 200.

Detailed Description Text (19):
The area specifications are organized hierarchically, each area specification containing one or more lower level area specifications, until the leaf area specifications are reached. Thus area specification 202 corresponds to the entire package 102 in image 100 and contains intermediate area specifications 206 and 218. Area specification 206 contains intermediate area specification 210 which contains leaf area specification 214 corresponding to all the small square areas (e.g. areas 120 and 122) in image 100.

Detailed Description Text (20):
The intermediate area specifications 206, 210 and 218 identify areas that are larger than the leaf areas 120 and 140 (corresponding to area specifications 214 and 224), but smaller than the complete package 102 (corresponding to area specification 202). Selecting appropriate intermediate area specifications for the model is an important step in the method of the invention.

Detailed Description Text (21):
The manner in which an image is hierarchically organized in the model and the intermediate area specifications and area specification pattern usage entities may be understood by considering the sample image in FIGS. 5(a)-5(d). In each of FIGS. 5(a)-(d), the drawing is divided between the left side and the right side. On the left of each drawing are one or more areas that have unique area specifications. On the right side is an image hierarchically organized with areas that match the area specifications on the left. As you proceed from FIG. 5(a) to FIG. 5(d) the area specifications on the left change from the lowest level (leaf area specifications) to the highest level (the entire area).

Detailed Description Text (31):
As can be seen by the paths leading from area specifications 210 and 218 to leaf area specification 214, information about an area type may be stored in a single location even though an area of that type is placed in different larger areas having different higher level area specifications.

Detailed Description Text (34):
In the first stage, referred to as "basic assembly", leaf areas are identified in the image, they are classified as being of one or more types accordingly to some characteristic of the leaf areas (relevant to an application), and adjacent leaf areas of equivalent type are grouped into one or more larger areas. These large areas containing identical adjacent leaf areas of a single type are referred to herein as "area arrays". Finally, area arrays that are of equivalent size and shape are identified. By identifying equivalent area arrays, only the unique area arrays need to be processed further. The results of the subsequent processing for each unique area array can then be directly applied to the corresponding equivalent area arrays.

Detailed Description Text (35):
In the second stage of the method, the unique area arrays are partitioned into areas, referred to herein as "partition areas". The first partition area corresponds to the entire area of the area array. Subsequent partition areas, if any, are smaller. The partitioning proceeds according to certain partitioning rules in an algorithm described below which creates patterns of smaller and smaller partition areas. When the same process is followed for area arrays of different shapes assembled from the same type of leaf area, the same patterns are often produced which results in common information storage about the partition areas when the area specification model is complete.

Detailed Description Text (36):
The partitioning is preferably done recursively with the first partitioning producing large partition areas and subsequent partitioning producing smaller

partition areas until the leaf area size has been reached. Unique partition areas are then identified and area specifications are generated for them. Area specifications are also generated for the areas corresponding to the unique area arrays and for the unique types of leaf areas.

Detailed Description Text (41):
one area specification for each unique type of leaf area.

Detailed Description Text (45):
b). identifying leaf areas in the image;

Detailed Description Text (46):
c). classifying the leaf areas as being of one or more unique types according to a characteristic of the leaf areas.

Detailed Description Text (47):
d). grouping adjacent leaf areas of equivalent type into one or more area arrays;

Detailed Description Text (51):
g). generating an area specification for an area corresponding to the entire image area, each unique partition area and each unique leaf area.

Detailed Description Text (53):
FIG. 6 provides one embodiment of the invention according to the above described method. A flat area representation of a packaging structure 602 with leaf areas is provided in image form in step 604. The leaf areas are of different types, according to the differences that are of importance to the application to use the model. Thus, different distinctions between areas may be made at this point to identify features that are relevant to the application that will use the model and ignore distinctions between areas that are not relevant.

Detailed Description Text (54):
In this embodiment of the method, the leaf areas are grouped into the area arrays in a two step process. First, in step 606, the image is scanned in one dimension to recognize leaf areas that are of equivalent type and are adjacent in the scanning direction. Second, in step 608, the image is scanned and grouped in the second dimension. The result of the two direction scanning is to identify all area arrays within the image composed of adjacent identical leaf areas. Steps 606 and 608 completely divide the image into mutually exclusive areas comprising the area arrays. The term "area array" is used because the two dimension scanning process produces areas composed of vertical columns and horizontal rows of leaf areas. Finally, equivalent area arrays, i.e., areas of identical size and shape or other unifying characteristics, are identified in step 610.

Detailed Description Text (55):
These basic assembly steps, following the above-described method, are graphically illustrated in FIG. 7. The flat image 702 includes two types of leaf areas: rectangular areas (e.g. 704) and square areas (e.g. 706). Area 708 is identical to area 704 except that it is turned 90.degree..

Detailed Description Text (56):
The results of the one dimensional (vertical) grouping of step 606 are shown in 710. During the vertical scan, adjacent leaf areas of identical type are grouped together into vertical groups as indicated by the different background markings in 710. The results of the second dimensional grouping in step 608 are illustrated in image 712.

Detailed Description Text (58):
Thus, upon completion of the basic assembly steps, all contiguous areas composed of identical leaf area types have been identified to form a set of one or more unique

area arrays. Each of the unique area arrays will be partitioned. Thus, in image 714 the upper area array 716 will be partitioned and the partitioning for that area array will also apply to area array 718 at the bottom of image 714.

Detailed Description Text (59):
Although the preferred method of identifying the unique area arrays is done as illustrated in FIGS. 6 & 7 with a two dimension scanning process, other methods may be used which identify adjacent leaf areas of equivalent type and assemble them into area arrays.

Detailed Description Text (61):
Starting with a unique area array of size N.sub.x by N.sub.y leaf areas, the primary constraints are:

Detailed Description Text (62):
1) a Desired Number Of Leaf Areas Range,

Detailed Description Text (64):
3) a Desired Minimum Number Of Children In One Direction At Leaf Level.

Detailed Description Text (69):
The Desired Number of Leaf Areas Range indicates the target number of leaf areas at the lowest level of the area hierarchy tree. This target can be supplied by the user based on prior experience or determined in a separate algorithm step that attempts to balance a desired height of the tree and a typical number of branches in the tree that optimizes the tree structure for typical searches. The number is derived by utilizing the total number of leaf areas and attempting to generate a tree that has a Depth (Height), defined as the distance between the root and the deepest leaf, that is determined internally to permit for tree searches of logarithmic orders of the numbers in the Desired Number of Children At Higher Levels Range. This type of operation is well known to those skilled in the art of algorithm development.

Detailed Description Text (70):
The invention targets to achieve popular low numbers (when the numbers for Desired Number of Children At Higher Levels Range numbers are not supplied) such as 4 (quad trees, etc.) so that the number of children at the highest level of the tree is small and, the lowest levels of the tree have a larger number of children before "leaf level".

Detailed Description Text (72):
The desired number of leaf areas can be derived from a Desired Area Range for different area leaf types when the Desired Number of Leaf Areas Range is not supplied or to find the best number of Leaf Areas when the Desired Area Range is supplied. If this range is not supplied, the invention will derive it in steps that follow the same procedure outlined for deriving the Desired Number of Leaf Areas Range. This time the calculation includes the actual size of the area in the calculation of the depth of the tree.

Detailed Description Text (74):
The derivation of the Desired Number of Leaf Areas Range and the Desired Area Range is performed concurrently when both ranges are not supplied. The Desired Minimum Number of Children in One Direction at Leaf Level is determined using the Desired Number of Leaf Areas Range and the Desired Area Range in combination with the Desired Aspect Ratio Range.

Detailed Description Text (79):
FIG. 8(a) illustrates the partitioning of the central area array in image 714 according to the area restructuring algorithm. The central area array is composed of a 4.times.8 array of leaf areas of type similar to area 706. In the first step,

the central area array 802 is split into four partition areas 804, 806, 808 and 810. In a second iteration of the restructuring algorithm area 804 is partitioned into a 2.times.4 array of leaf areas of the type identical to area 706.

Detailed Description Text (82):
1. If (N.sub.x *N.sub.y) is in the Desired Number of Leaf Areas Range then Exit.

Detailed Description Text (92):
2.1 Determine New Direction of Partitioning (the direction with N.sub.x or N.sub.y) such that any N.sub.x or N.sub.y that is below the Desired Minimum Number of Children in One Direction at Leaf Level will not be partitioned.

Detailed Description Text (94):
2.2.1 Divide the Number of Leaf Areas in that Direction by (2 to Maximum of the Desired Minimum Number of Children At Higher Levels Range).

Detailed Description Text (101):
FIG. 8(a) illustrates the result of the partitioning process on the central region of the image. In the first step of the recursive partitioning process, the partition criteria includes a Desired Minimum Number Of Children In One Direction At Leaf Level that included the numbers 4, 6 and 8. The constraint Desired Number Of Children At Higher Levels Range is 2-4. FIG. 8(b) shows the corresponding expanded hierarchical area structure. Note that the area made of 2.times.4 leaf areas are the same across the entire package thereby minimizing the overall number of area specifications. The information about these areas can be shared.

Detailed Description Text (103):
The rectangular nodes (e.g. 918, 920) correspond to multiple areas, including all the partition areas resulting from the recursive area restructuring algorithm down to the leaf areas. The multiple individual nodes to which the rectangular nodes correspond are shown in FIG. 9(b). For example, rectangular node 930 includes nodes 950 and 952. Node 950 corresponds to area 804 while node 952 corresponds to one of the 8 leaf areas within area 804.

Detailed Description Text (107):
As occurs in every hierarchical area model according to the present invention, there is a single top area specification corresponding to the entire area (1102) and one area specification at the bottom for each unique type of leaf area (2 area specifications (1114 and 1116).

Detailed Description Text (108):
Note that at two locations within the hierarchical area specification model 1100 separate branches of the model merge into a single area specification. This occurs at area specification 1114, which is a leaf area specification, and at area specification 1134. This merger indicates that identical leaf areas or identical intermediate areas are found in two different area arrays and the information that is common is stored in a single location.

Detailed Description Text (109):
The final post-processing stage of the method is shown in FIGS. 10 and 11(b). FIG. 10 corresponds to FIG. 9(a) except that two new nodes 1002 and 1004 have been introduced into the area tree. It may occur that there are a large number of area specifications at the top of the tree when built based on the partitioning steps previously described. Therefore, in order to meet the partitioning criteria fully, the post-processing stage performs a grouping of the high level area to reduce the number of children of the root area by introducing additional area specifications at the top. FIG. 10 illustrates the situation of post-processing grouping to satisfy the given Desired Minimum Number Of Children In One Direction At Leaf Level criteria.

Detailed Description Text (113):
1. Label Temporarily all Children of Top Level of the Area Tree as belonging to one fictitious Simulated Leaf area type. (The object of this labeling will be clear below.)

Detailed Description Text (114):
2. Determine the New Desired Leaf Area Assembly Range as equal to the Desired Number of Children At Higher Levels Range.

Detailed Description Text (115):
3. Perform all Basic Assembly Steps on the Simulated Leaf areas of the Top Level of the Area Tree in a fashion similar to that performed originally for the actual leaf areas. The only difference now is to eliminate redundancy in the Horizontal and vertical scan such that Simulated Leaf elements do not get included in more than one array area.

Detailed Description Text (117):
Treat Simulated Leaf areas similar to real leaf areas except when performing the steps of creating new area specifications and when determining whether the operation minimizes the number of area specifications. In those steps, the Simulated Leaf areas are operated upon using their actual original area specifications.

CLAIMS:

1. A method for restructuring a physical design image into a data model comprising the steps of:

providing a physical design image;

identifying leaf areas in the physical design image;

classifying the leaf areas as being of one or more types according to a characteristic of the leaf areas;

grouping adjacent leaf areas of equivalent type into one or more area arrays;

recognizing equivalent adjacent and non-adjacent area arrays and forming a set of one or more unique area arrays;

partitioning each unique area array into one or more partition areas;

recognizing equivalent partition areas and forming a set of one or more unique partition areas; and

generating area specifications for each unique partition area and for each unique leaf area.

2. A method for restructuring an image into a data model according to claim 1 wherein the step of grouping adjacent leaf areas of equivalent type into one or more area arrays comprises the steps of:

scanning the image in a first direction;

grouping leaf areas of equivalent type that are adjacent in the first direction to form one dimensional area arrays;

scanning the image in a second direction; and

grouping equivalent one dimensional area arrays that are adjacent in the second

direction to form the area arrays.

9. A method for restructuring an image into a data model according to claim 5 wherein the number of different types of partition areas are reduced by identifying identical patterns of leaf areas to reduce the number of different area specifications.

10. A method for restructuring an image into a data model according to claim 1 wherein the step of identifying leaf areas includes the step of flattening the hierarchy and creating leaf areas if the image has been previously organized into a hierarchy.

11. A method for restructuring an image into a data model according to claim 1 wherein the step of classifying the leaf areas as being of one or more types ignores different characteristics between areas provided that such areas have common characteristics relevant to an application.

13. A method for restructuring an image into a data model comprising the steps of:

providing an image;

identifying leaf areas in the image;

classifying the leaf areas as being of one or more types according to a characteristic of the leaf areas;

grouping adjacent leaf areas of equivalent type into one or more area arrays;

recognizing equivalent area arrays and forming a set of one or more unique area arrays;

recursively partitioning each unique area array by desired partitioning criteria to form a hierarchical array of one or more partition areas according to a sequence of steps including:

(a) exiting the sequence of steps if an area of N.sub.x by N.sub.y leaf areas has a number of leaf areas within it that is within a predetermined Desired Number of Leaf Areas Range;

(b) creating a set of area slices of size New N.sub.x, by New N.sub.y leaf areas by slicing the area of N.sub.x by N.sub.y leaf areas;

(c) recognizing equivalent slices from the set of area slices and removing redundant slices to form a set of unique area slices;

(d) identifying slices from the set of unique area slices that have previously been assigned an area specification and removing such slices from the set of unique area slices; and

(e) for every remaining distinct slice in the set of unique area slices performing the steps of:

(i) creating a new area specification, and

(ii) repeating the sequence of steps substituting New N.sub.x and New N.sub.y for N.sub.x and N.sub.y ;

recognizing equivalent partition areas and forming a set of one or more unique partition areas; and

generating area specifications for each unique partition area and for each unique leaf area.

15. A method for restructuring an image into a data model according to claim 14 wherein the direction of the slices is determined by a partitioning criterion equal to Desired Minimum Number of Children in One Direction at Leaf Level, the slicing being performed in a direction such that N.sub.x or N.sub.y will not be sliced if below the Desired Minimum Number of Children in One Direction at Leaf Level.

Previous Doc Next Doc Go to Doc#

L17: Entry 1 of 5 File: USPT Feb 17, 2004

DOCUMENT-IDENTIFIER: US 6694323 B2
TITLE: System and methodology for providing compact B-Tree

Abstract Text (1):
An improved method for creating an index based on a path-compressed binary trie in a database system comprising database tables and indexes on those tables is described. For a given index to be created, a path-compressed binary trie for the given index is determined. The path-compressed binary trie comprises internal nodes and leaf nodes. Based on a traversal of the path-compressed binary trie, an index is created comprising a first array of internal nodes encountered during the traversal, and a second array of leaf nodes encountered during the traversal. The database system employs said first and second arrays for providing index-based access for a given key value.

Brief Summary Text (13):
For enhancing the speed in which the DBMS stores, retrieves, and presents particular data records, the DBMS usually maintains one or more database indexes on a database table. A database index, typically maintained as a B-Tree (or B+-Tree) data structure, allows the records of a table to be organized in many different ways, depending on a particular user's needs. An index may be constructed as a single disk file storing index key values together with unique record numbers. The index key values are a data quantity composed of one or more fields from a record which are used to arrange (logically) the database file records in some desired order (index expression). The record numbers are unique pointers or identifiers to the actual storage location of each record in the database file. Both are referred to internally by the system for locating and displaying records in a database file.

Brief Summary Text (14):
Searching for a particular record in a B-Tree index occurs by traversing a particular path in the tree. To find a record with a particular key value, one would maneuver through the tree comparing key values stored at each node visited with the key value sought. The results of each comparison operation, in conjunction with the pointers stored with each node, indicate which path to take through the tree to reach the record ultimately desired. Ultimately, a search will end at a particular leaf node which will, in turn, point to (i.e., store a pointer to or identifier for) a particular data record for the key value sought. Alternatively, the leaf nodes may for "clustered indexes" store the actual data of the data records on the leaf nodes themselves.

Brief Summary Text (17):
Many B-Tree variants have been introduced in the literature and in practice. These can be classified according to the technique used for searching within a page. Most implementations are comparison-based in which searching is typically done by performing a binary search on a sorted array of keys. A variety of optimizations have been developed to improve upon this basic scheme. For an overview of such optimizations, see e.g., Graefe, G. and Larson, P. A., "B-Tree Indexes and CPU caches," in 17th International Conference on Data Engineering (ICDE), pages 349-358, Washington-Brussels-Tokyo, April 2001, published by IEEE Computer Society Press.

Brief Summary Text (18):
Radix-based B-Tree variants are less common, with the majority intended for text indexing, not on-line transaction processing (OLTP). These radix-based variants can be further categorized as to whether they manipulate the on disk representation directly or not. Examples in the first group include the string B-Tree (see e.g., Ferragina, P. and Grossi, R. "The string B-Tree: A new data structure for string search in external memory and its applications," Journal of the ACM, 46(2): 236-280, 1999) and the string R-Tree (see e.g., Jagadish, H. V., Koudas, N., and Srivastava, D., "On effective multi-dimensional indexing for strings," in volume 29, pages 403-414 of the Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Tex.). They store a compressed linearization of the search structure on disk and reconstruct it on demand. A disadvantage of this approach, particularly in an OLTP environment, is that it complicates buffer management: the reconstructed search structure likely will not fit on a single page. Further, this approach does not solve the problem of constructing a cache-efficient in-memory representation. An example of a radix-based B-Tree variant that manipulates the on-disk representation directly is Ferguson's Bit-Tree (see e.g., Ferguson, D. "Bit-Tree, a data structure for fast file processing," Communications of the ACM, 35(6): 114-120, 1992). Other examples in this group include the pkB-Tree (see e.g., Bohannon, P., McIlroy, P., and Rastogi, R., "Main-Memory index structures with Fixed-Size partial keys," volume 30, 2 of SIGMOD Record, pages 163-174, ACM Press), a Bit-Tree refinement suitable for main memory databases.

Brief Summary Text (19):
Although B-Tree indexes are widely used to improve DBMS performance, they add overhead to the DBMS as a whole and, therefore, these indexes need to be carefully structured and used to maximize system performance, especially when indexing long values. One approach to indexing long values is to store partial key information: keys are represented with a small normalized prefix together with an identifier (ID) of the row containing the key (if needed). If the prefix alone is not sufficient to resolve a comparison, a full compare is performed against the values in the underlying row, an expensive proposition if a cache miss is incurred. While this implementation works surprisingly well for many indexes--trading off the occasional full compare for better fanout--there is room for improvement, especially if the schema is not carefully designed. Further, even with this type of implementation, index size can still be an issue in resource-constrained environments, such as a typical Windows CE environment.

Brief Summary Text (24):
One of the most widely used indexing schemes in database systems is the B-Tree index scheme (including implementation variants such as a B+-Tree) in which the keys or keywords are kept in a balanced tree structure and the lowest level (i.e., leaf nodes) of the tree points at the data records, or, in some cases, contains the actual data records themselves. General techniques for the construction and operation of B-Trees are well documented in the technical, trade, and patent literature. For a general description, see Sedgewick, R., "Algorithms in C--Third Edition," Addison-Wesley, 1998. For a survey of various B-Tree implementations, see Comer, D., "The Ubiquitous B-Tree," Computing Surveys, Vol. 11, No. 2, June 1979, pp. 121-137. Unless otherwise specified, in this document the term B-Tree includes B-Trees, B+-Trees, and the like.

Brief Summary Text (26):
In a database system, an index is a list of keys or keywords which enable a unique record to be identified. Indexes are typically used to enable specific records to be located more rapidly as well as to enable records to be more easily sorted (e.g., sorted by the index field used to identify each record).

Brief Summary Text (30):

A Patricia tree or path-compressed binary trie is a search tree in which each non-leaf node includes a bit offset and has two children. A Patricia tree is based upon a trie structure with each node including the index of the bit to be tested to decide which path to take out of that node. The typical way that a Patricia tree is searched is to start at the root node and the tree is traversed according to the bits of the search key. For example, if a given bit offset is `0`, then the search proceeds to the left child of the current node. If the bit offset is `1`, then the search proceeds to the right child of the current node. For further information on Patricia trees, see Morrison, D., "PATRICIA--Practical Algorithm to Retrieve Information Coded in Alphanumeric," Journal of the ACM, 15(4): 514-534, 1968, the disclosure of which is hereby incorporated by reference.

Brief Summary Text (36):
A trie is a binary tree that has keys associated with each of its leaves defined recursively as follows: the trie for an empty set of keys is a null link; the trie for a single key is a leaf containing that key; and the trie for a set of keys of cardinality greater than one is an internal node with a left link referring to the trie for the keys whose initial bit is 0 and right link referring to the trie for the keys whose initial bit is 1, with the leading bit considered to be removed for purposes of constructing the subtrees. For a description of tries, see e.g., Sedgewick, R., "Algorithms in C--Third Edition", Addison-Wesley, 1998, the disclosure of which is hereby incorporated by reference. Searching a trie typically proceeds from the root to a leaf, where the edge taken at each node depends on the value of an attribute in the query. Typical trie implementations have the advantage of being fast, but the disadvantage of achieving that speed at great expense in storage space. For further information regarding trie structures, see e.g., Comer, D., "Heuristics for Trie Index Minimization," ACM Transactions on Database Systems, Vol. 4, No. 3, September 1979, pages 383-395, the disclosure of which is hereby incorporated by reference.

Brief Summary Text (38):
An improved method for creating an index based on a path-compressed binary trie in a database system comprising database tables and indexes on those tables is described. For a given index to be created, a path-compressed binary trie for the given index is determined. The path-compressed binary trie comprises internal nodes and leaf nodes. Based on a traversal of the path-compressed binary trie, an index is created comprising a first array of internal nodes encountered during the traversal, and a second array of leaf nodes encountered during the traversal. The database system employs said first and second arrays for providing index-based access for a given key value.

Drawing Description Text (12):
FIG. 7B illustrates the bit offsets and keys associated with internal nodes and leaf nodes of the ptree of FIG. 7A.

Detailed Description Text (25):
For enhancing the storage, retrieval, and processing of data records, the server 330 maintains one or more database indexes 345 on database tables 350. These database indexes 345 facilitate quick access to the data records. A database index, typically maintained as a B-Tree data structure, allows the records of a table to be organized in many different ways, depending on a particular user's needs. An index may be constructed as a single disk file storing index key values together with unique record numbers. As previously described, an index key value is a data quantity composed of one or more fields from a record which are used to arrange (logically) the database file records by some desired order (index expression). The record numbers are unique pointers to the actual storage location of each record in the database file. Both are referred to internally by the system for locating and displaying records in a database file. Of particular interest at FIG. 3, the indexes 345 may include both conventional comparison-based B-Trees as well as Patricia tree-based B-Trees constructed in accordance with the present invention

which are referred to herein as "compact B-Trees." These compact B-Trees are Patricia tree-based (or trie-based) indexes constructed in accordance with the present invention as hereinafter described in detail.

Detailed Description Text (26):
As clients insert more and more data into the table(s) 350, the indexes 345 continue to grow. Two parts of the server 330 play a role in the processing and maintenance of the indexes: the access methods 370 and the DB store 375. The DB store 375, which includes a page and index manager module, includes interfaces (or submodules) for the following index operations: create, insert, delete, reorganize, and search. The create submodule is used to create an index (e.g., bulk load operation). The insert submodule is used for inserting an index entry, while the delete submodule is used for deleting an index entry. The reorganize module is used to improve the space utilization of an index by packing the index entries more tightly. The search submodule functions to find a particular key value in a tree. The B-Tree insert, delete and search operations can be viewed as starting with a B-Tree search, starting from the root node or page of the B-Tree. Searching for a particular record in the B-Tree occurs by traversing a path in the B-Tree, searching each page encountered for the key value sought. The insert submodule serves to insert a new entry into a B-Tree. Once the leaf level is reached, if enough space does not exist (e.g., for insertion), then a B-Tree split routine is invoked for splitting a B-Tree page. In the currently preferred embodiment, splits propagate from the top down. Traversal operations in the currently preferred embodiment also proceed from top to bottom. The system and method for providing the compact B-Trees of the present invention will now be described.

Detailed Description Text (29):
B-Trees are fundamental to the maintenance of indexes. FIG. 4 illustrates a simple B-Tree 400, which comprises a root node 411, internal nodes 421, 422, and leaf (terminal) nodes 431, 432, 433, 434. As shown, therefore, a B-Tree consists of a plurality of nodes arranged in a tree. Each node may, in turn, be thought of as a block of records. As shown by the root node 411, each node stores one or more key values ("keys") together with pointers to children nodes (e.g., nodes 421, 422 for root node 411).

Detailed Description Text (30):
Searching for a particular record in the B-Tree occurs by traversing a particular path in the tree. To find a record with a particular key value, one would maneuver through the tree comparing key values stored at each node visited with the key value sought. The results of each comparison operation, in conjunction with the pointers stored with each node, indicate which path to take through the tree to reach the record ultimately desired. Ultimately, a search will end at a particular leaf node, such as leaf node 431. The leaf node will, in turn, point to (i.e., store a pointer to or identifier for) a particular data record for the key value sought. Alternatively, the leaf nodes may for "clustered indexes" store the actual data of the data records on the leaf nodes themselves.

Detailed Description Text (32):
The present invention comprises a system and method providing a compact, Patricia tree-based B-Tree index. In a compact B-Tree index constructed in accordance with the present invention, each page of the B-Tree contains a local Patricia tree instead of the usual sorted array of keys. The search structure used may also be described as a "path-compressed binary trie." For further information on Patricia trees, see Morrison, D., "PATRICIA--Practical Algorithm to Retrieve Information Coded in Alphanumeric," Journal of the ACM, 15(4): 514-534, 1968. Also see e.g., Nilsson, S. and Tikkanen, M., "Implementing a dynamic compressed trie," in Proceedings Workshop on Algorithm Engineering (WAE) 1998, Saarbrucken, Germany, August 20-22, pages 341-350. Following the literature, this document refers to this path-compressed binary trie as a "Patricia tree." Utilization of a Patricia tree representation provides a B-Tree variant that, while compact, is efficient to

manipulate directly and provides significant space and performance benefits over existing B-Tree indexes. The following discussion first generally describes Patricia trees. The methodology of the present invention for providing compact B-Trees is then described.

Detailed Description Text (33):
The present invention comprises a new B-Tree variant referred to herein as a "compact B-Tree" which is designed to improve the space and time performance of indexes. The compact B-Tree indexes of the present invention are based on a radix-based search, designed so that references to indirectly stored keys are likely not to incur additional cache misses in the common case. For information on radix-based search, see, e.g., Sedgewick, R., "Algorithms in C--Third Edition", Addison Wesley, 1998. For additional information regarding radix-based routines, see e.g., commonly-owned U.S. Pat. No. 5,924,091 titled "Database system with improved methods for radix sorting." While it is natural to expect good relative performance for longer keys, experience has shown that this approach is competitive even for small key sizes.

Detailed Description Text (37):
The following discussion uses as an example a table consisting of a set of N records, each of the form r.sub.i =(k.sub.i, .alpha..sub.i) in which k.sub.i is called the key, .alpha..sub.i the associated information, and i the record identifier or rid. An index on the table provides an ordering on the rids i.sub.1, i.sub.2, . . . , i.sub.N such that k.sub.i1.ltoreq.k.sub.i2.ltoreq. . . . .ltoreq.k.sub.iN. The index operations of interest are:

Detailed Description Text (39):
2. Keys

Detailed Description Text (40):
Without loss of generality, it is assumed that the keys k.sub.i appearing in an index are unique. If not, the rid of the record that the key appears in can always be appended (i.e., an index formed on (k.sub.i, i)). This approach is appropriate in any case since the appropriate index entry will need to be updated when a particular row is updated or deleted. See e.g., Graefe and Larson "B-Tree indexes and CPU caches," above.

Detailed Description Text (41):
It is also assumed that all keys can be normalized to binary strings in an order preserving fashion. Therefore, the focus of the present discussion is restricted to indexing these strings. One consequence of this assumption is that the indexes described in this document are not able to index all possible database values. In particular, it is not always feasible to construct an order preserving normalization for Java programming language values.

Detailed Description Text (42):
Further, it is assumed that the normalization is such that no key is a prefix of another. This is trivially so for fixed length keys. For variable length keys an end marker needs to be added in an order preserving fashion. Conceptually, all keys are padded with binary `1`s to M bits, where M is greater than the length in bits of any key that could possibly be encountered. Binary `1`s are added instead of `0`s because this gives a convenient representation for a value greater than any naturally occurring key. This simplifies the algorithms allowing keys to be padded for alignment at will.

Detailed Description Text (43):
To summarize, in the following discussion it is assumed that a key k is a binary string of the form b.sub.1 b.sub.2 . . . b.sub.M. The term k[i] will be used for b.sub.i and k[i:j] for b.sub.i, b.sub.i+1, . . . b.sub.j. A key k.sub.1 is less than a key k.sub.2 if there exists a j such that k.sub.1 [1:j-1]=k.sub.2 [1:j-1]

and k.sub.1 [j]<k.sub.2 [j].

Detailed Description Text (46):
A B-Tree page is either a leaf page or an internal page. Internal pages contain pointers to child pages (page ids or pids), leaf pages contain pointers to records (record ids or rids). Each pointer in a B-Tree page has an associated key. The key associated with record id i is just k.sub.i. B-Tree pages can be represented as illustrated in FIG. 5A. FIG. 5A illustrates a B-Tree page 510 in which K.sub.1 <K.sub.2 < . . . <K.sub.n and Pi points to a subtree with rows containing keys in the range (K.sub.i-1, K.sub.i). Typical B-Tree implementations omit the last key from nonleaf pages. In the currently preferred embodiment, the last key is retained in order to simplify the implementation at negligible cost. While the keys in leaf pages must be row values, various key values may be used for the internal pages, so long as they satisfy the above criteria.

Detailed Description Text (48):
For a description as to how these index operations can be implemented in terms of the page level operations, see e.g., Sedgewick, R., "Algorithms in C++," Addison-Wesley, 1998, or Knuth, D., "Sorting and Searching," volume 3 of The Art of Computer Programming, Addison-Wesley, Reading, Mass., USA, second edition, 1998. When the search structure consists of a sorted list of keys, these page level operations are trivial. However, this is no longer the case when the search structure is a Patricia tree.

Detailed Description Text (51):
Generally, a Patricia tree or path-compressed binary trie is a search tree in which each non-leaf node includes a bit offset and has two children. The typical way that a Patricia tree is searched is to start at the root node and examine the bit at the specified offset to determine if it is zero (`0`) or one (`1`). If the bit offset is `0`, then the search proceeds to the left child of the current node. If the bit offset is `1`, then the search proceeds to the right child of the current node. When the next node is visited, the bit at the offset specified in this node is examined. Based upon whether the bit being examined is a `0` or `1`, the search proceeds to the left or to the right. The search continues in this fashion until a leaf is reached.

Detailed Description Text (52):
It is convenient to describe Patricia trees recursively. A Patricia tree is either a leaf L(k) containing a key k or a node N(d, l, r) containing a bit offset d>0 along with a left subtree l and right subtree r. All leaves descended from N(d, l, r) must agree on the first d-1 bits; all leaves descended from the left child l must have a `0` as the dth bit and all leaves descended from the right child r must have a `1` as the dth bit. This definition is similar to that used by Sedgewick ("Algorithms in C," above), but differs from that of Knuth ("Sorting and Searching," above) in that the absolute bit offset is stored in the node rather than the number of bits skipped. A leaf of a Patricia tree is identified with the key it holds.

Detailed Description Text (56):
Observation 3. A leaf a is to the left of a leaf b if the key in the leaf a is less than the key in the leaf b.

Detailed Description Text (57):
Observation 4. The offset of the first bit at which two keys differ is the bit offset found in their lowest common ancestor. The importance of these observations to the methodology of the present invention is described below.

Detailed Description Text (59):
Given the key k.sub.i stored in a leaf L(k.sub.i), one can find the leaf by doing a blind search from the root node. If a node N(d, l, r) is encountered, the search

continues on to l if k.sub.i [d]=0 and r otherwise. FIG. 5C illustrates a blind search for the node 1110 in the Patricia tree 520 of FIG. 5B. The examined bits of node 1110 are underlined at FIG. 5C. (2) Insert

Detailed Description Text (60):
To insert a value k.sub.i, a blind search is performed to find a leaf L(k.sub.j). The key k.sub.j agrees with k.sub.i on all of the bits tested on the path from the root to k.sub.j, but k.sub.i must differ from k.sub.j at some bit position since all keys are distinct. Let d be the first differing bit position. If a node N(d', l', r') is encountered during the blind search with d'>d, then let the subtree s be the first such node; otherwise, let s=L(k.sub.j). If k.sub.i [d]=0, then insert a new node N(d, L(k.sub.i), s); otherwise, insert a node N(d, s, L(k.sub.i)). FIG. 5D illustrates the insertion of a node 1100 into the Patricia tree of FIG. 5C. The added elements are bolded in the Patricia tree 530 at FIG. 5D. (3) Search

Detailed Description Text (61):
For a search for a given key k, one would like to find the leaf f=L(k.sub.j) such that k.sub.j is the smallest key where k.sub.j.gtoreq.k. This requires an approach similar to that described above for the insertion operation. Let d and s be determined as above. If k[d]=0, then f is the leftmost leaf of s as in FIG. 5D. Otherwise, f is the leaf immediately to the right of (the rightmost leaf of) the subtree s. Searching for 1010 also finds 1110 by first doing a blind search to 1000, then determining that f is the leaf immediately to the right of 1000, namely 1110. (4) Delete

Detailed Description Text (62):
Given a key k, the delete operation finds and removes L(k) from the tree. If the tree consists only of L(k), then the result is the empty tree. Otherwise, there is either a node n=N(d, L(k), s) or n=N(d, s, L(k)). Delete consists of removing node n and replacing it with s. (5) Splitting and Merging

Detailed Description Text (65):
The merge operation is the inverse of the split operation. Given two Patricia trees T.sub.l and T.sub.r, with T.sub.l to the left of T.sub.r (i.e., if k.sub.l is the rightmost key of T.sub.l and k.sub.r is the leftmost key of T.sub.r, then k.sub.l <k.sub.r), the merge operation forms the Patricia tree containing the combined set of keys. It merges the paths to k.sub.l and k.sub.r such that the bit offsets along the resulting path are in increasing order. It proceeds until it arrives at subtrees l and r in T.sub.l and T.sub.r, respectively, that are either leaves or are rooted by a node containing a bit offset greater than the offset d of the first bit at which k.sub.l and k.sub.r differ. At this point a new node N(d, l, r) is inserted. This approach guarantees that the bit offsets along the merged path are distinct since all of the keys in T.sub.l are less than all of the keys in T.sub.r.

Detailed Description Text (67):
Any representation of a Patricia tree must encode the bit offsets found in the internal nodes and the shape of the tree. The present invention provides a method for recovering the shape of a Patricia tree from the offsets alone, if these offsets are listed as encountered in an in-order traversal of the tree. The only information required to merge two Patricia trees whose ordering is known (i.e., it can be determined which tree is the left and which is the right) is the offset of the first bit in which the neighboring keys differ. Recall from observation 4 above that the offset of the first bit at which two keys differ is the bit offset found in their lowest common ancestor. Thus, the list of bit offsets encountered in an in-order traversal of a Patricia tree is simply a list of the offsets at which consecutive keys first differ. Viewing each leaf as a separate subtree, the original Patricia tree can be reconstructed simply by repeatedly merging the leaves until the reconstruction process results in a single tree.

Detailed Description Text (68):
This representation has been previously described in the literature (see e.g., Ferguson. "Bit-tree, a data structure for fast file processing," above). However, this correspondence has been overlooked. Given this correspondence, it is less surprising that searching a Bit-Tree requires only one key access (see e.g., Bohannon, P., McElroy, P., and Rastogi, R., "Main-Memory index structures with Fixed-Size partial keys," volume 30, 2 of SIGMOD Record, pages 163-174, ACM Press). Ferguson implements the above-mentioned Patricia tree operations directly on the list of distinction offsets. Of these, only the implementation of search is non-trivial and requires a scan of the entire page. While this is reasonable for small page sizes (such as found in main memory databases, where the B-Tree nodes are roughly the size of a cache line) it impacts performance when the page size is large.

Detailed Description Text (69):
The compact B-Tree of the present invention provides an improved Patricia tree encoding that trades off compactness for speed of search. This compact B-Tree structure uses a Patricia tree within each page. It should be noted that the compact B-Trees of the present invention are not of a fixed order and, therefore, should not be confused with those of Rosenberg and Snyder (see e.g., Rosenberg, A., and Snyder, L. "Time- and space-optimality in B-Trees," ACM TODS, 6(1): 174-193, 1981). The following discussion describes the compact B-Tree of the present invention in greater detail.

Detailed Description Text (71):
1. Representing Keys

Detailed Description Text (72):
The idea of using Patricia trees within B-Tree pages requires a decision about how to represent the keys, and how to encode the Patricia tree. Recall that the Patricia tree stores keys for each of the leaf nodes. A naive implementation of this approach is unlikely to do better than a comparison based approach. As long as the ordering of the keys can be recovered, a search can be performed in logarithmic time without any recourse to the Patricia tree. Therefore, the additional bookkeeping that would be required to maintain a Patricia tree in this situation is avoided.

Detailed Description Text (73):
Instead of storing the keys in the pages of the index, the method of the present invention is to instead store a pointer to the key, which is stored on another disk page. One potential issue with this approach is that de-referencing the pointer may cause a cache miss, thereby resulting in an input/output (I/O) page read latency. One full key comparison is needed per level of the index; if each of these required waiting for a read request, then the compact B-Tree index would behave substantially worse than existing implementations.

Detailed Description Text (74):
One approach to avoid this cost is to try to avoid full comparisons. This is the approach taken by pkB-Trees (see e.g., Rosenberg and Snyder, "Time-and space-optimality in B-Trees," above), in which a small fragment of each key is stored with each leaf. Usually, this fragment is enough to avoid following a pointer to do a full comparison operation. Although this approach does avoid many full comparison operations, it does so at the expense of increasing the information stored in each node, and thus decreasing the fan out.

Detailed Description Text (75):
Another approach is to try and place keys on pages that are likely to already be in the cache or on pages that will likely be needed in the near future. In either of these cases, the extra cache miss resulting from following the key pointer is eliminated. Instead of using extra overhead to store an additional key pointer for

each leaf in the Patricia tree, it is natural to re-use the pointers that are already stored in the tree. These pointers are descendant pointers for internal pages and rids for leaf pages.

Detailed Description Text (77):
For leaf pages, keys are stored in the associated rows. A successful search will do a full compare with the matching row. If values need to be retrieved from the matching row in the near future, the full compare likely does not impose an additional cache miss since it prevents a cache miss when reading the row. In the currently preferred embodiment, the matching row is retrieved immediately after searching in the index. This characteristic means that successful searches never cause an additional cache miss doing a full compare with the matching row in the database.

Detailed Description Text (79):
In addition to the issue of how to store keys in the index pages, the representation of Patricia trees must also be determined. The disk-based representation should be compact in order to achieve good fan-out. However, it is also desirable to have logarithmic search behavior on average. If one was willing to settle for linear search behavior, the Bit-Tree encoding described above is a good alternative. Given that the shape of a Patricia tree is determined by the data, linear behavior cannot be avoided in all cases. For example, the set of keys k.sub.i,where k.sub.i [j]=0 for all j but i, produces a right-deep tree.

Detailed Description Text (81):
FIG. 6 illustrates an exemplary compact B-Tree index page 600 constructed in accordance with the present invention. As shown, the method of the present invention is to store each Patricia tree as an ordered array of leaves together with an array that encodes the structure of the internal nodes of the Patricia tree. The leaf array simply holds the bit C.sub.i and the pointers P.sub.i, where C.sub.i is the bit that indicates the next key is equal in value to this key and P.sub.i is the pointer to the key value. The bit C.sub.i is maintained only for leaf pages. The node array stores the bit offset D and number of leaves L in the left subtree as encountered in a pre-order traversal. In addition to the two arrays, each page also stores a header block, the total number of leaves N, and a key chosen by the page splitting algorithm. Note that the key, if it exists, is the value associated with the pointer to this page in the parent page. The key is an upper bound for the values referenced by the P.sub.i and a strict lower bound for the values on the page to the right. As described below in the discussion regarding splitting, the lower bound requirement can be relaxed for pages that are rightmost children.

Detailed Description Text (82):
One advantage of this representation is that one can trace the path from the root to the ith leaf simply with the knowledge of i. In the currently preferred embodiment, the header is 30 bytes, N takes 4 bytes, D.sub.i and L.sub.i are 2 bytes each, and the leaf array entries (C.sub.i P.sub.i) are 4 bytes for internal pages and 5 bytes for leaf pages. The maximum fan-out for internal pages is given by 1/8 (page size-key size-34). For example, with a 4K page size and 200 byte key, the internal pages can have a fan-out of 482 and leaf pages can have a maximum fan-out of 429. This compares favorably with existing B-Tree implementations with the same parameters.

Detailed Description Text (85):
In the above example, {x} denotes a pointer to key x. While it is possible to reconstruct the tree, it is more efficient to manipulate the representation directly. This avoids the memory management problems as well as the administrative cost associated with expanding the tree. In order to manipulate the compact encoding directly, the operations on Patricia trees need to be defined to operate instead on the encoding.

Detailed Description Text (88):
Performing a blind search on the encoding for a key K proceeds by keeping track of the current node j and, for the subtree rooted at j, the leftmost leaf i and number of leaves n it contains. When visiting an internal node j, the search proceeds to the left if the D.sub.j th bit in K is `0` and to the right otherwise.

Detailed Description Text (89):
As previously described, a blind search of a Patricia tree or path-compressed binary trie structure typically starts at the root node with an examination of the bit at the specified offset to determine if it is zero (`0`) or one (`1`). Based upon whether the bit being examined is a `0` or `1`, the search proceeds to the left or to the right. This process continues until a leaf node is reached. This is a radix-based approach in which bits are examined rather than the whole key comparisons which are typically used in conventional B-Tree index schemes. (2) Search

Detailed Description Text (90):
Searching the encoding for a key K consists of first performing a blind search which finds leaf k in the manner described above. One then compares K to the key K.sub.j which is pointed to by pointer P.sub.k using a function D (k.sub.1, k.sub.2) that returns the offset d at which two keys k.sub.1 and k.sub.2 differ, as well as k1[d]. (Note that if k.sub.1 =k.sub.2, M and 1 are returned). If the two keys match, then in fact the correct leaf was chosen (this is always the case for successful searches). Otherwise, the appropriate leaf that is greater than K must be found. This is done this by retracing the path followed by the blind search until locating a leaf or a node whose bit offset is greater than d, the offset of the first bit at which K and the K.sub.j differ. There is no need to inspect any bits of either K or K.sub.j in the second pass: the leaf index provides enough information to recover the path. The following second pass proceeds in a manner similar to the blind search code described above, except that k, the index of the searched-for leaf relative to the first leaf in the current subtree, is also maintained.

Detailed Description Text (91):
In other words, when the leaf node is reached through a blind search, the search may or may not have reached the correct value. When the search reaches a leaf node, it is known that the key being searched for and the key associated with the leaf agree at the bits that have already been examined. However, the two keys could differ at one of the unexamined bit positions. Accordingly, the key associated with the leaf is examined to determine if it matches the search value. If it does not match the search value, the first bit at which the two keys differ is determined. At this point, the search backs up to the point at which a bit of the search value differs from that of the leaf node offset value. Essentially, the information that is gathered during the comparison of values at the leaf node indicates the correct leaf to visit. Also note that CPU cache misses are not likely to be encountered when retracing the path followed by the blind search as shown above.

Detailed Description Text (97):
When splitting a page in two, a value is selected that separates the resulting pages. This value is inserted into the parent. Ideally this value should be as short as possible. Note that, as discussed above, the bit offsets in the tree are just the lengths of values needed to discriminate between adjacent leaves. In the currently preferred implementation, the split operation is given a pair of indexes (i, j) with i<j. The split operation is allowed to split the page at any point in this range; the point chosen is the one that minimizes the resulting key length.

Detailed Description Text (98):
Note that the key associated with the rightmost pointer of a non-leaf page can be any value larger than the largest value on the page (in particular the value

consisting of M ones which has a zero length representation). This value can be changed after the split of an internal page by deleting the old rightmost value and replacing it with the shorter one. The specific steps involved in a split operation in the currently preferred embodiment will now be described. (4) Merge

Detailed Description Text (101):
The most common index operation is to search for a given value. If the index is not unique, then a range query must be done. One simple approach would be to find the first and one past the last index entries containing the value by supplying `-.infin.` and `+.infin.` for the rid, respectively, and then return the intervening entries as the normalization provides for strict upper and lower bounds for each column. However, this approach is too expensive, especially if only a few values are returned and the cost of the second probe is significant. The approach of the present invention is to tag the pointers in a leaf page with a bit which indicates that the next row contains a key which differs from the current key only in the rid. This bit avoids the retrieval of a non-matching row to stop the scan and saves one row access (and possible page read) for each such range scan. Furthermore, it is not expensive to maintain this bit during insert operations on the Patricia tree. For deadlock avoidance, each leaf page also stores the rid of the first row on the next leaf page. Since the leaf pages are doubly linked, this does not introduce extra update overhead, although it does require an additional eight bytes for each leaf page.

Detailed Description Text (104):
The methodology of the present invention may be illustrated by example. The following discussion will describe several operations on an exemplary compact B-Tree index constructed in accordance with the present invention. For purposes of explanation, a tree structure is used to illustrate several operations on exemplary path-compressed binary tries (Patricia trees) which, in accordance with the present invention, are used within B-Tree pages. However, it should be emphasized that the actual operations are performed on the above-described array representations of these Patricia trees, enabling these operations to be handled more efficiently.

Detailed Description Text (106):
2. Bit Offsets and Keys Associated with Nodes

Detailed Description Text (107):
In accordance with the present invention, each internal node x of a ptree has an associated bit offset (bit(x)) and each leaf node y has an associated key (key(y)). FIG. 7B illustrates a ptree 720 including the bit offsets and keys associated with internal nodes and leaf nodes of the ptree 710 of FIG. 7A. As shown at FIG. 7B, each internal node of the ptree of FIG. 7A is labeled with the bit offset (bit(x)) and each leaf node with the key (key(y)). All keys beneath a node x have the same initial bit(x)-1 bits prefix(x). The keys in the left subtree have a subsequent `0` bit, while those in the right subtree have a subsequent `1` bit. For example, as shown at FIG. 7B the keys in subtrees E and F agree on the first four bits (prefix (d)=`0100`) but differ in the subsequent bit. The difference is that the next bit in the keys in subtree E is a `0` while the next bit in the keys in subtree F, which is to the right, is a `1`.

Detailed Description Text (109):
FIG. 7B also includes a high-level array representation 725 of the exemplary ptree 720. As shown, the array representation 725 contains two arrays: a first array (leaf array) 726 for leaves, and a second array (internal array) 727 for internal nodes. Each of these arrays lists the nodes as encountered in pre-order traversal of the ptree 720. Note that for subtrees, the label of the subtree is used to represent the list of nodes in the array representation of the subtree (possibly empty in the case of a leaf). In the leaf array, the key (key(y)) is stored for each leaf y. In the internal array 727, each node x contains the number of leaves in the left subtree of node x and the bit offset associated with node x (i.e.,

(each node x stores (leaves (left(x)), bit(x)) where leaves (y) and left(y) represent the number of leaves beneath and the left subtree of node y, respectively). As shown at FIG. 7B, the leaf array 726 of the array representation 725 is simply the list of (pointers to) keys in ascending order.

Detailed Description Text (117):
FIG. 7C illustrates a blind search operation on the ptree array representation 725 shown at FIG. 7B. The blind search for the key value `01001` commences at the root node with i=1, j=1, and n=4 (the number of leaves in ptree 720). As shown at 1, the second bit of `01001` is `1`, so the search proceeds to the right. At this point i is increased to 2 (increased by L value of 1), j is also increased to 2, and n is decreased to 3. Note that as the search proceeds to the right, leaves in the left subtree are skipped. As shown at 2, the next bit examined is a `0`, so the search proceeds to the left, with i remaining at the same value (2), j increased from 2 to 3, and n decreased to 2. The next bit examined at 3 is a `1` so the search proceeds to the right. At this point, the value of n is equal to 1 as the leaf has been reached and the key value (`01001`) located.

Detailed Description Text (119):
The following describes the process for a search operation (search(v)) which returns i such that key(i-1)<v<=key(i). In other words, the search operation finds the least i such that key(i)>=v). The steps involved in this search operation are as follows: 1. l<-blindsearch(v). Blind search for key value(v) to reach leaf node l. 2. Calculate the bit offset d as the first bit at which the search value(v) and the key of the leaf node (key(l)) differ. If they do not differ (i.e., if key of leaf node matches search value), return l as correct value has been located. 3. Trace path to the lth leaf, stopping if reaching an internal node j with a bit offset greater than the calculated offset d (i.e., stopping if bit(j)>d) or stopping if l is reached. 4. Return i if v[d]=0, or i+n if v[d]=1 (where i and n are the state variables from tracing the path to l (i.e., i is the leftmost leaf of n leaves under j). For example, suppose in the tree shown in FIG. 7A bit(d) is greater than bit(c)+1. Then searches for v=prefix(c) 0 and w=prefix(c) 1 will both stop before node d. However, search(v) will return the number of leaves in subtree B as shown at FIG. 7A, while search (w) will return the number of leaves in subtree B, plus those in subtrees E and F (i.e., the number of leaves under internal node d as shown at FIG. 7A).

Detailed Description Text (121):
FIGS. 7D-E illustrate two slightly different insert operations performed on a ptree. The initial ptree before either of these two insert operations is performed is the same ptree 710 previously shown at FIG. 7A. As previously described, ptree 710 includes leaves B, E, F, and g as well as internal nodes a, B, c, d, E, and F. The process for inserting y with a key key(y) into the ptree 710 of FIG. 7A is as follows: 1. l<-blindsearch(y). Blind search for key value(y) to reach leaf node l. 2. Calculate the bit offset d of first bit at which the key(y) and the key of the leaf node (key(l)) differ. If they do not differ, raise an error condition, as key values must be unique. 3. Trace path to the lth leaf, stopping if visiting an internal node j with a bit offset greater than the calculated offset d (i.e., stopping if bit(j)>d) or stopping if arriving at l. While tracing the path, whenever traversing to the left, 1 is added to the value of leaves (left(j)), where j is the parent node, as a leaf will be inserted in the left subtree of j. 4. Insert y in the leaf array at i and, depending upon whether y[d] is `0` or `1`, insert (i, d) or (n, d) in the internal array at j (where i and n are the state variables from tracing the path to l (i.e., i is the leftmost leaf of n under j).

Detailed Description Text (122):
FIGS. 7D-E illustrate the ptree after insertion of node y. As described above, after the initial blind search, a bit offset d is computed as first bit at which the search value(y) and the key of the leaf node (key(l)) found as a result of the blind search differ. For purposes of this example, the bit offset in node d is

greater than the bit offset in node c plus one (i.e., bit(d)>bit(c)+1). FIG. 7D illustrates a ptree 730 after insertion of a node y having a key value key(y) =prefix(c) 0 into the ptree 710 of FIG. 7A. The inserted nodes are indicated by dashed lines in ptree 730 at FIG. 7D. FIG. 7E represents a ptree 740 after insertion of a node y having a different key value into the same ptree 710 of FIG. 7A. FIG. 7E shows the different ptree resulting from the insertion of a node having a key value of key(y)=prefix(c) 1.

Detailed Description Text (123):
As shown at FIG. 7D, if key(y)=prefix(c) 0, a new internal node x is inserted below and to the left of node c. As also shown, the leaf array 735 includes nodes/subtrees B, y, E, F, and g, while the internal array includes nodes/subtrees a, B, c', x, d, E, and F. In this situation, c'=(leaves (left(c)+1, bit(c)) and x= (1, bit(c)+1).

Detailed Description Text (124):
Alternatively, if key(y)=prefix(c) 1, a new internal node x is inserted beneath node c, node y is inserted below and to the right of this internal node x and internal node d and subtrees E and F are to the left of node y, all as illustrated at FIG. 7E. The nodes/subtrees stored in the leaf array include B, E, F, y, and g. The internal array includes nodes/subtrees a, B, c', x, d, E, F, where c'=(leaves (left(c))+1, bit(c)) and x=(number of leaves in subtrees E and F, bit(c)+1).

Detailed Description Text (126):
The following describes a delete operation for deletion of node(v) of a ptree. The process involved in deletion of an exemplary node v is as follows: 1. l<- blindsearch(v). Blind search for key value(v) to reach leaf node l. 2. If the search value(v) and the key of the leaf node (key(l)) differ, raise an error condition, as v is not a key in this situation. 3. Trace path to the ith leaf, subtracting one from leaves (left(j)), where j is the parent node, when traversing left in the tree, as a leaf will be removed from node j's left subtree. 4. Delete the ith entry from the leaf array and the j th entry from the internal array, where j is the parent of l (i.e., the last internal node visited before arriving at leaf l).

Detailed Description Text (137):
In the currently preferred embodiment, the same application-programming interface provides access to both conventional (comparison-based) B-Tree indexes as well as to the trie-based compact B-Tree indexes of the present invention. When an index is to be created, at step 803, a determination is made as to which type of index should be created. If the width of the columns being indexed (i.e., key length) falls within a pre-determined range, then compact B-Tree indexes are used as these compact B-Tree indexes typically provide better performance than comparison-based (conventional) B-Tree indexes in these circumstances. Otherwise, conventional B-Tree indexes are generally used in circumstances where the indexes have either very short keys or very long keys. For purposes of the following discussion, it is assumed that the index to be created is a compact B-Tree index which is to be created in accordance with the methodology of the present invention.

Detailed Description Text (138):
At step 804, the index creation submodule of the system of the present invention is called to create a compact B-Tree index (e.g., to create an index on a particular column of a database table). The index creation submodule receives a handle to the index to which the values are to be appended. Each of the individual values that are to be inserted into the index during index creation is appended to the end of the index. At this point the meta data necessary for creating the index is also generated. At step 805, a loop is established to walk through (i.e., process) each of the records in the table. At step 806, as each row of the table is accessed, the key value of the column being indexed is retrieved. In the currently preferred embodiment, these key values are in sorted order as a result of the prior sorting

operation at (optional) step 802 above. Alternatively, if the key values are not already in sorted order, they would be sorted before they are appended to the index as described below.

Detailed Description Text (140):
With a conventional (comparison based) B-Tree index, the key values are simply appended to the end of an array. However with the present Patricia tree-based index (i.e., compact B-Tree), at step 809 the correct location for insertion of the key value must be determined before the element is inserted and the update is made. Appending a key value is a simpler case than an inserting a value(as described above) as when appending a key value one knows to always follow the right most spine of the index (i.e., the search is always going right). In addition, the value of the right-most leaf node of each page (i.e., the current largest value on each page at each level) is tracked to optimize the insertion of values into the index. When a value is to be inserted, the value is compared to this tracked value to determine the first bit at which the values differ. The values will differ because if the values would not differ, the method of the present invention provides for appending the row ID to ensure that values can be distinguished. If the value is greater than the previous entry, there will be a `1` bit on (rather than a `0`) in the value that is to be entered. As a result, the location for insertion of the entry can thus be determined by examining a bit beyond the first bit at which the values differ. Once the location for insertion of an entry is determined, at step 810 the value is inserted into the Patricia tree-based index and the update is made. The key value that is inserted is inserted as a new right sub-tree and the values that are below the position reached in the tree will be the left sub-tree below this position.

Detailed Description Text (141):
It should be noted that the above discussion refers to insertion of an element in a tree structure. However, the method of the present invention provides for the use of an array-based representation of the Patricia tree-based index as previously shown at FIG. 6 and at FIGS. 7A-H. Instead of storing some sort of tree structure with links, two arrays are used as described above. The first array (or leaf array) is an array of leaves which are stored in order. The second array (or internal array) is an array of internal nodes. As previously described, each internal node consists of a bit offset and the number of leaves in the left sub-tree. The tree is traversed in a set order and mapped to these arrays with the result that the position in the array is related to the traversal order. In addition, the array contains sufficient information which enables a tree representation to be reconstructed from the array representation.

Detailed Description Text (143):
Like the compact B-Tree of the present invention, the Bit-Tree of Ferguson may be used for OLTP workloads. Both manipulate compact Patricia tree representations directly. However, the compact B-Tree implementation differs from the Bit-Tree of Ferguson in that the same representation is used for the internal pages, with keys being stored in the associated child page, while the Bit-Tree reverts to a conventional representation for internal pages. Bit-Tree leaf pages are more compact; the compact B-Tree of the present invention requires an additional 2 bytes per entry for internal nodes (the size of the left subtree of the node) as well as room for a key. However, these additional bytes enable a compact B-Tree page to be searched in logarithmic time whereas the Bit-Tree leaf pages can only be searched in linear time.

Detailed Description Text (144):
The pkBit-Tree of Bohannon is an optimization of the Bit-Tree for main memory databases (see e.g., Bohannon, et. al "Main-Memory index structures with Fixed-Size partial keys," above). The pkB-Tree generalizes and improves upon Bit-Tree by storing a small amount of key information to avoid full compares. While searching is still linear, extra information added to the node (typically 2 to 4 bytes)

allows many full compares to be avoided. A potential disk cache miss is hidden by fact that the referenced page is either likely already in cache or it will likely be fetched in the near future. As with the present invention, the pkB-Tree internal and leaf pages share a common representation. However, the pkB-Tree is a B-Tree, not a B+-Tree. As a result, keys in internal pages reference the rows in the underlying table. This requires an extra rid per internal page entry.

Detailed Description Text (145):
The simplified String B-Tree described in the literature also requires two entries per child. See e.g., Ferragina, P. and Grossi, R., "Fast string searching in secondary storage: Theoretical developments and experimental results," In Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA '96), pages 373-382, Atlanta, June 1996, ACM Press. See also, Ferragina, P. and Grossi, R., "The string B-Tree: A new data structure for string search in external memory and its applications," Journal of the ACM, 46(2): 236-280, 1999. However, as with the pkB-Tree, the simplified String B-Tree is a B-Tree, not a B+-Tree, and uses the same representation for internal and leaf pages. A significant advantage of the simplified String B-Tree for long strings is that it avoids rescanning: bits in the searched key are looked at only once, not once per level. Unlike the previously mentioned implementations, the simplified String B-Tree does not manipulate the on disk representation directly; rather it decompresses and compresses it as required.

Detailed Description Text (147):
One Patricia tree representation that addresses some these issues is the Index Fabric, a stratified Patricia tree (see e.g., Cooper, B., Sample, N., Franklin, M., Hjaltason, G., and Shadmon, M., "A fast index for semistructured data," In Proceedings of the Twenty-seventh International Conference on Very Large Databases, Roma, Italy, 11-14 Sep., 2001, pages 341-350, Los Altos, Calif. 94022, USA, 2001, Morgan Kaufmann Publishers. Conceptually, this approach starts with a Patricia tree and carves out page size subtrees. These subtrees are then indexed by another Patricia tree. The process is repeated until one is left with a root Patricia tree that fits in single page. An advantage of this representation is that one is almost certain that the page de-referenced for a full compare will be the next page visited. Another advantage is that rescanning can be avoided. However, implementing range queries (not described (or required) by Cooper, et. al., above) will be considerably more complex. In general, it is not possible to order two leaf pages, since the keys on one page could be intermingled with the keys on the other page.

Detailed Description Text (148):
Under most circumstances the performance of the compact B-Tree indexes of the present invention are comparable to that of existing B-Tree indexes if no full comparisons are required. The advantage of these compact B-Tree indexes depends on the size of the normalized key: if the key requires a full 10 bytes, then the compact B-Tree indexes have a slight advantage. However, for short keys, existing B-Tree indexes have a slight advantage. In addition, when very little cache memory is available (for example, less than 0.5% of the sum of the index and table sizes) compact indexes are at the mercy of the buffer manager: if the upper levels of the index are not kept in memory the compact index performance can be degraded. The performance of the compact B-Tree indexes is relatively independent of key length: compared to existing indexes, it effectively eliminates the full compare overhead. This full compare overhead typically results in about a 25% performance penalty.

Detailed Description Text (149):
A key advantage of compact B-Tree indexes is their size. Compact B-Trees are roughly half the size of the existing B-Tree indexes. This space savings results from the significantly smaller per-leaf overhead of the compact B-Tree indexes. For compact B-Trees, the per-leaf overhead is 8 bytes for internal nodes and 9 bytes for leaf nodes, independent of the size of the normalized keys. For existing B-Tree index implementations, this overhead is usually between 12 and 41 bytes for

internal nodes and between 8 and 37 bytes for leaf nodes, depending on the length of the normalized keys and the length of the normalized prefix stored in each page. Typical overheads are 19 for internal nodes and 15 for leaf nodes. While the per-leaf overhead is lower for compact B-Trees, they do require up to 200 bytes per page to store a single normalized key. For most indexes, this is more than offset by the savings due to lower per-leaf overhead.

Detailed Description Text (150):
The use of compact B-Trees in the database management system of the currently preferred embodiment is limited to cases where the normalized key is no more than 200 bytes and the index page size is at least 2,000 bytes. These restrictions ensure that no more than ten percent of each index page is needed for key storage. In practice, keys are typically much smaller than this 200 byte limit.

Detailed Description Paragraph Table (4):
header 3 {1000} {1100} {1111} [2 1] [4 1] key

Current US Original Classification (1):
707/101

Current US Cross Reference Classification (1):
707/102

CLAIMS:

1. In a database system comprising database tables and indexes on those tables, an improved method for creating an index based on a path-compressed binary trie, the method comprising: for a given index to be created, determining a path-compressed binary trie for the given index, said path-compressed binary trie comprising internal nodes and leaf nodes; based on a traversal of said path-compressed binary trie, creating an index comprising: a first array of internal nodes encountered during the traversal, and a second array of leaf nodes encountered during the traversal; and wherein said system employs said first and second arrays for providing index-based access for a given key value.

2. The method of claim 1, wherein each particular internal node includes a bit offset indicating a particular bit value of the given key value to be examined at the particular internal node during traversal of the index.

10. The method of claim 7, further comprising: receiving a request to search the index based on a particular key value; while traversing the first array, using said information about subtree size to skip internal nodes in the first array that can be excluded, based on key value, from the search of the index.

11. The method of claim 1, further comprising: receiving a request to search the index based on a given key value; traversing internal nodes in the first array to locate a particular leaf which may contain said given key value; comparing said given key value to a key value at the particular leaf.

12. The method of claim 11, further comprising: if said given key value does not match the key value at the particular leaf, locating the position of a leaf containing a value matching said given key value based, at least in part, on the first bit at which said given key value and the key value at the particular leaf differ.

13. The method of claim 1, further comprising: receiving a request to insert a given key value into the index; traversing internal nodes in the first array to locate a particular leaf in the second array for insertion of said given key value; comparing said given key value to a key value at the particular leaf; and determining a position for insertion said given key value based, at least in part,

on the first bit at which said given key value and the key value at the particular leaf differ; and inserting said given key value at the determined position and adjusting nodes in the first array and second array based upon insertion of said given key value.

14. The method of claim 1, further comprising: receiving a request to delete a particular key value; traversing internal nodes in the first array to locate said particular key value in the second array; and adjusting nodes in the first array and second array based upon deletion of said particular key value.

15. The method of claim 1, wherein said determined path-compressed binary trie comprises a tree-based structure in which each leaf node contains key values and each internal node includes a bit offset and has two child subtrees.

16. The method of claim 15, wherein said bit offset at a given internal node indicates a bit value of a particular key value to be examined to determine a direction to take at the internal node in traversing the path-compressed binary tree to locate said particular key value.

20. The method of claim 1, wherein said determined path-compressed binary trie comprises a tree-based data structure, wherein each particular internal node stores information indicating a size for at least one of its subtrees and a bit offset indicating a bit value to be examined to determine a direction to take in traversing the tree-based data structure to locate a given key value.

22. The method of claim 1, wherein said second array includes a pointer to a key value for each leaf node.

25. The method of claim 1, further comprising: for each leaf node, storing in said second array an indicator as to whether the next key value is equal to the key value of the current leaf node.

28. In a database system, said database system storing a plurality of data records, an improved method for creating a path-compressed binary trie index of such records, the method comprising: adding key values from at least some of the data records to at least one index page; for each index page, determining a path-compressed binary trie of key values in said index page, said path compressed binary trie including leaf nodes and non-leaf nodes; traversing said path-compressed binary trie and creating a path-compressed binary trie index by performing the substeps of: for each non-leaf node of said path-compressed binary trie, storing in a first array a bit offset and information about size of at least one subtree of said each non-leaf node, said bit offset indicating a particular bit value of the given key value to be examined at the non-leaf node during traversal of the index; and for each leaf node of said path-compressed binary trie, storing in a second array a pointer to a key value.

31. The method of claim 28, further comprising: for each leaf node, storing in said second array an indicator as to whether the next key value is equal to the key value of the current leaf node.

33. The method of claim 28, wherein said key values are stored on child pages for non-leaf index pages.

34. The method of claim 28, wherein said key values are obtained from the data records for leaf index pages.

36. The method of claim 28, further comprising: for at least some index pages, storing a key value for the index page, said key value associated with a pointer to the index page in the parent of the index page.

37. The method of claim 28, further comprising: sorting said key values before creating said path-compressed binary trie index.

40. The method of claim 38, wherein said determining step includes evaluating the length of the key values.

46. The method of claim 28, further comprising: receiving a request to search the index based on a particular key value while searching pages during index traversal, while traversing the first array, using said information about subtree size to skip non-leaf nodes in the first array that can be excluded, based on key value, from the search of the index.

48. The method of claim 28, wherein said step of storing said pointer in a second array includes storing a key value on child pages for non-leaf index pages.

49. The method of claim 28, further comprising: normalizing said key values to binary strings in an order preserving fashion.

50. The method of claim 28, wherein said pointers to key values comprise data record identifiers for leaf index pages.

52. The method of claim 28, further comprising: for each leaf node, storing an indicator in said first array as to whether the key value associated with the next leaf node is equal to the key value of the current leaf node.

53. The method of claim 28, wherein said step of adding key values includes linking a particular index page to its parent page and allocating a new index page when said particular index page is full.

55. The method of claim 28, further comprising: appending a row identifier to said key values in the event key values are not unique.

Previous Doc Next Doc Go to Doc#

L19: Entry 1 of 1 File: USPT Jan 6, 2004

DOCUMENT-IDENTIFIER: US 6675163 B1
TITLE: Full match (FM) search algorithm implementation for a network processor

Abstract Text (1):
Novel data structures, methods and apparatus for finding a full match between a search pattern and a pattern stored in a leaf of the search tree. A key is input, a hash function is performed on the key, a direct table (DT) is accessed, and a tree is walked through pattern search control blocks (PSCBs) until reaching a leaf. The search mechanism uses a set of data structures that can be located in a few registers and regular memory, and then used to build a Patricia tree structure that can be manipulated by a relatively simple hardware macro. Both keys and corresponding information needed for retrieval are stored in the Patricia tree structure. The hash function provides an n.fwdarw.n mapping of the bits of the key to the bits of the hash key. The data structure that is used to store the hash key and the related information in the tree is called a leaf. Each leaf corresponds to a single key that matches exactly with the input key. The leaf contains the key as well as additional information. The length of the leaf is programmable, as is the length of the key. The leaf is stored in random access memory and is implemented as a single memory entry. If the key is located in the direct table then it is called a direct leaf.

Brief Summary Text (9):
The routing of the receive packet is based on the accompanying address string. The address string is used as a search key in a database which contains the address string along with other pertinent details such as which router is next in a delivery of a packet. The database is referred to as a routing table, while the link between the current router and the next router is called the next hop in the progress of the packet. The routing table search process depends on the structure of the address as well as the organization of the tables. For example, a search key of a size less than 8 bits and having a nonhierarchal structure would most efficiently be found in a routing table organized as a series of address entries. The search key would be used as an index in the table to locate the right entry. For a search key of a larger size, say thirty-two bits, the corresponding routing table may have more than 10,000 entries. Organizing the database as a simple table to be searched directly by an index would waste a large amount of memory space, because most of the table would be empty.

Brief Summary Text (12):
The second step, that of determining the direction of the destination network, is not usually performed by a linear search through a table since the number of network addresses would make such a table difficult to manage and use. In the prior art, when address strings conformed to the three-level hierarchy of network address, subnet address and host identification, routers performed the determination using one of several well-known techniques, such as hashing, Patricia-tree searching, and multilevel searching. In hashing, a hash function reduces the network portion of the address, producing a small, manageable index. The hash index is used to index a hash table and to search for a matching hash entry. Corresponding to each hash entry of the hash table is the address of an

output interface pointing in the topological direction of a corresponding network. If a match is found between the hash network portion and a hash entry, the message is directed towards the corresponding interface and destination network.

Brief Summary Text (14):
Patricia-tree searching avoids the collisions encountered by hashing methods. This method of searching requires that all address strings and accompanying information, such as related route information, be stored in a binary tree. Starting from the most significant bit position within the address string, the search process compares the address, bit by bit, with the tree nodes. A matched bit value guides the search to visit either the left or the right child node and the process is repeated for the next bit of the address. The search time is proportional to the size of the longest address string stored. In Patricia-tree searching, the difference between the average search time and the worst case search time is not very large. In addition, the routing table is organized quite efficiently. It requires less memory than comparable routing tables of hashing methods. Patricia-tree searching handles the worst case searches better than the hashing methods, but in most cases it takes significantly longer to locate a match. Therefore, many conventional routers use a combination of hashing and Patricia-tree searching. This combination is called multilevel searching.

Brief Summary Text (15):
Multilevel searching joins hashing with Patricia-tree searching. A cache stores a hash table containing a subset of the most recently, and presumably most commonly, routed network addresses, while a Patricia-tree stores the full set is of network addresses. As the message is received, the destination address is hashed onto the table. If it is not located within a pre-determined period of time, the address is passed to the Patricia-tree search engine which insures that the address, if stored, will be found.

Brief Summary Text (16):
In the prior art, there are a number of known tree search algorithms including fixed match trees, longest prefix match trees and software managed trees. Fixed match trees are used for fixed size patterns requiring an exact match, such as layer 2 Ethernet MAC tables. Longest prefix match trees are used for variable length patterns requiring only partial matches, such as IP subnet forwarding. Software managed trees are used for patterns that are defined as ranges or bit masks, such as filter rules. In general, lookup is performed with the aid of a tree search engine (TSE).

Brief Summary Text (18):
It is an object of this invention to provide for the implementation in hardware of a Full Match tree search algorithm for Patricia trees. It describes how the memory structures are set up so that they can serve the purpose of the algorithm, and how the hardware processes these structures.

Brief Summary Text (20):
The main concept is that a key is input, a hash function is performed on the key, a direct table (DT) is accessed, and the tree is walked through pattern search control blocks (PSCBs) and ends up with a leaf.

Brief Summary Text (21):
The problem solved is the design of a set of data structures that can be located in a few registers and regular memory, and then used to build a Patricia tree structure that can be manipulated by a relatively simple hardware macro. In the Patricia tree, both keys and corresponding information needed for retrieval are stored.

Brief Summary Text (22):
The key is the information that is to be searched on and matched. Initially, the

key is placed in a register and hashed. The result is the hash key and the actual search will happen on the hash key. The hash function could be the null hash, and then the hash key will be exactly the same as the key. The hash function provides an n.fwdarw.n mapping of the bits of the key to the bits of the hash key.

Brief Summary Text (23):
The data structure that is used to store the hash key and the related information in the tree is called a leaf. Retrieving the leaf is the purpose of this algorithm. Each leaf corresponds to a single key that matches exactly with the input key. In this implementation the leaf contains the key, and appended to it is the additional information to be stored. The length of the leaf is programmable, as is the length of the key. The leaf is stored in random access memory and is implemented as a single memory entry. If the key is located in the direct table (DT) then it is called a direct leaf.

Drawing Description Text (7):
FIG. 5 illustrates a tree data structure for the full match search algorithm in accordance with a preferred embodiment of the present invention.

Drawing Description Text (9):
FIG. 7 illustrates the effect on exemplary data structures of having direct leaves enabled in accordance with a preferred embodiment of the present invention.

Drawing Description Text (10):
FIG. 8 illustrates an exemplary structure of a DT entry and pattern search control block (PSCB) line formats in a Full Match search tree in accordance with a preferred embodiment of the present invention.

Drawing Description Text (11):
FIG. 9 illustrates an example of a search using a Full Match search in accordance with a preferred embodiment of the present invention.

Drawing Description Text (12):
FIG. 10 illustrates the processing logic of the Full Match (FM) search algorithm in accordance with a preferred embodiment of the present invention.

Drawing Description Text (15):
FIG. 13 illustrates the fixed leaf format for FM trees.

Drawing Description Text (16):
FIG. 14 illustrates an exemplary architecture for a tree search engine in accordance with a preferred embodiment of the present invention.

Detailed Description Text (3):
Up to N parallel protocol processors are available. In an embodiment of 16 protocol processors, 16,384 words of internal picocode instructions store and 32,768 words of external picocode instructions store are available to provide 2,128 million instructions per second (MIPS) of aggregate processing capability. In addition, each protocol processor has access to M hardware accelerator coprocessors which provide high speed pattern search, data manipulation, internal chip management functions, frame parsing, and data prefetching support. In a preferred embodiment control storage for the protocol processors is provided by both internal and external memories: 32 K of internal static random access memory (SRAM) 28 for immediate access, external zero bus turnaround (ZBT) SRAM 30 for fast access, and external double data rate (DDR) dynamic random access memory (DRAM) 32 for large storage requirements.

Detailed Description Text (9):
FIG. 3 illustrates an exemplary embodiment of a protocol processor. The coprocessors associated with each of the programmable protocol processors 40

provide the following functions: 1. A data store coprocessor 64 interfaces frame buffer memory 42, 44 (ingress and egress directions) to provide direct memory access (DMA) capability; 2. A checksum coprocessor 62 calculates header checksums; 3. An enqueue coprocessor 66 controls access to the 256-bit working register, containing key frame parameters. This coprocessor interfaces with the completion unit 46 to enqueue frames to the switch and target port queues; 4. An interface coprocessor provides all protocol processors access to internal registers, counters and memory for debug or statistics gathering; 5. A string copy coprocessor enables efficient movement of data within the EPC; 6. A counter coprocessor manages counter updates for the protocol processors 40; 7. A policy coprocessor examines flow control information and checks for conformance with pre-allocated bandwidth.

Detailed Description Text (10):
Hardware accelerators 48 perform frame forwarding, frame filtering, frame alteration and tree searches. Other features incorporated into the network processor include innovative filter rule processing, hash functions and flow control.

Detailed Description Text (11):
The protocol processors 40 can enforce one hundred or more frame filter rules with complex range and action specifications. Filtering is essential for network security, and network processor hardware assists 48 provide wirespeed enforcement of these complex rule sets. Filter rules can deny or permit a frame or allocate quality of service (QoS) based on IP header information. Control point software for preprocessing rules automatically corrects logic errors. After a logically correct rule set has been entered, keys are formed from packet header information and are tested at wirespeed using the network processor's software managed trees.

Detailed Description Text (12):
Geometric hash functions exploit statistical structures in IP headers to outperform ideal random hashes. Consequently, the low collision rates enable high speed look-ups in full match tables without additional resolution searches.

Detailed Description Text (13):
Operating in parallel with protocol processor execution, the tree search engine 70 performs tree search instructions (including memory read, write or read-write), memory range checking and illegal memory access notification. FIG. 14 illustrates an exemplary embodiment of a tree search engine.

Detailed Description Text (16):
Data frames are received from the media by the PMM 22 and transferred to the data storage buffers 42. The PMM also performs CRC checking and frame validation during the receive process. The dispatcher 50 sends up to 64-bytes of frame information to an available protocol processor 40 for frame look-ups. The classifier hardware assist 48 supplies control data to identify frame formats. The protocol processor 40 uses the control data to determine the tree search algorithm to apply including fixed match trees, longest prefix match trees, or software managed trees.

Detailed Description Text (17):
Look-up is performed with the aid of a tree search engine (TSE) 70. The TSE 70 performs control memory 72 accesses, enabling the protocol processor 40 to continue execution. The control memory 72 stores all tables, counters and any other data needed by the picocode. For efficiency, a control memory arbiter 52 manages control memory operations by allocating memory cycles between the protocol processors 40 and a variety of on-chip and off-chip control memory options 54.

Detailed Description Text (22):
Egress tree searches support the same algorithms as are supported for ingress searches. Look-up is performed with the TSE 70, freeing the protocol processor 40 to continue execution. All control memory operations are managed by the control

memory arbiter 52, which allocates memory access among the processor complexes.

Detailed Description Text (24):
The tree search engine (TSE) 70 as depicted in FIG. 14 uses the concept of trees to store and retrieve information. Retrieval, i.e., tree-searches as well as inserts and deletes are done based on a key, which is a bit-pattern such as, for example, a MAC source address, or the concatenation of an IP source address and an IP destination address. An exemplary tree data structure 100 for use in the present invention is depicted in FIG. 5. Information is stored in a control block called a leaf 116, 118, 120, 122, which contains at least the key 102 (the stored bit pattern is actually the hashed key 106). A leaf can also contain additional information such as aging information, or user information, which can be forwarding information such as target blade and target port numbers. The format of a leaf is defined by picocode; the object is placed into an internal or external control store.

Detailed Description Text (25):
The search algorithm for trees operates on input parameters including the key 102, performs a hash 104 on the key, accesses a direct table (DT) 108, walks the tree through pattern search control blocks (PSCBs) 110, 112, 114 and ends up at a leaf 116, 118, 120, 122. Each type of tree has its own search algorithm causing the tree-walk to occur according to different rules. For example, for fixed match (FM) trees, the data structure is a Patricia tree. When a leaf has been found, this leaf is the only possible candidate that can match the input key 102. A "compare at the end" operation compares the input key 102 with the pattern stored in the leaf. This verifies if the leaf really matches the input key 102. The result of this search will be success (OK) when the leaf has been found and a match has occurred, or failure (KO) in all other cases.

Detailed Description Text (26):
The input to a search operation contains the following parameters: key The 176 bit key must be built using special picocode instructions prior to the search or insert/delete. There is only one key register. However, after the tree search has started, the key register can be used by the picocode to build the key for the next search concurrently with the TSE 70 performing the search. This is because the TSE 70 hashes the key and stores the result in an internal 192 bit HashedKey register 106. key length This 8 bit register contains the key length minus one bit. It is automatically updated by the hardware during the building of the key. LUDefIndex This is an 8 bit index into the lookup definition table (LUDefTable), which contains a full definition of the tree in which the search occurs. The internal structure of the LUDefTable is illustrated in FIG. 11. TSRNr The search results can be stored either in 1 bit Tree Search Result Areas TSR0 or TSR1. This is specified by TSRNr. While the TSE is searching, the picocode can access the other TSR to analyze the results of a previous search. color For trees which have color enabled (specified in the LUDefTable), the contents of a 16 bit color register 124 is inserted in the key during the hash operation.

Detailed Description Text (27):
For FM trees, the input key will be hashed into a HashedKey 106, as shown in FIG. 4. There are several fixed algorithms available. The algorithm that will be used is specified in the LUDefTable.

Detailed Description Text (28):
The lookup definition table is the main structure which manages tree search memory. The LUDefTable is an internal memory structure and contains 128 entries for creating trees. The LUDefTable contains entries that define the physical memory the tree exists in (e.g., DRAM, SRAM, internal RAM), whether caching is enabled, the size of the key and leaf, and the type of search action to perform. The LUDefTable is implemented as three separate random access memories--one RAM that is accessible only by the general processor tree handler (GTH) and two RAMs that are duplicates

of each other and are accessible by all picoprocessors.

Detailed Description Text (29):
The output of the hash function 104 is always a 176-bit number which has the property that there is a one-to-one correspondence between the original input key 102 and the output of the hash function 104. As will be explained below, this property minimizes the depth of the tree that starts after the direct table 108.

Detailed Description Text (31):
Colors can be used to share a single direct table 108 among multiple independent trees. For example, one use of a color could be a VLAN ID in a MAC source address (SA) table. In this case, the input key 102 would be the MAC SA, and the color 124 would be the VLAN ID (since the VLAN ID is 12 bits, four bits of the color would be unused, i.e., set to zero). After the hash function 104, the pattern used is 48+16=64 bits. The color is now part of the pattern and will distinguish between MAC addresses of different VLANs.

Detailed Description Text (33):
The first structure that implements the tree is called the direct table (DT) 108. Each entry in a DT table with N elements corresponds to a key whose first log.sub.2 N bits are the same as the index of that entry in the DT table, in binary form. For example, the 5.sup.th entry in an 16 entry DT table would correspond to keys whose first 3 bits are "0101". If there are no leaves that correspond to a key with the first log.sub.2 N bits the same as the index in the DT, then that entry is marked as empty. If there is only a single leaf that matches those bits, then inside that entry there is a pointer to a leaf. This pointer is the address in the memory that the leaf is stored. If there is more than one leaf that corresponds to keys with the same first bits, then the DT entry points to a PSCB structure 110, and also contains the next bit(s) to test (NBT) field 126. These two structures will be described below.

Detailed Description Text (34):
The DT table 108 is implemented in memory, and its size (length) and starting point are programmable. Another programmable feature is the use of what are called direct leaves. Instead of having the DT entry point to a leaf, which then must be read afterwards, the leaf can be stored in the location of the DT entry. This is called a direct leaf. The problem with this is, of course, a tradeoff in speed with the use of more memory for the DT entry. The memory size (its width) must be enough to accommodate a leaf, and not all of the DT entries will have leaves stored in them. However, a good hash function of the key could result in most of the leaves being attached to a single DT entry, so the speed tradeoff could be big.

Detailed Description Text (35):
In summary, a DT entry can be empty. In this case, no leaves are attached to this DT entry. The DT entry can point to a single leaf attached to this DT entry. In this case, the DT entry can point to a pattern search control block (PSCB) and also contain the next bit(s) to test (NBT) for that PSCB. There is more than one leaf attached to this DT entry. Finally, the DT entry can contain a direct leaf.

Detailed Description Text (36):
A PSCB represents a branch in the tree. In the preferred embodiment there is a 0-branch and a 1-branch. The number of branches emanating from a PSCB is variable depending on the number of bits used to designate the branches. If n bits are used, then 2.sup.n branches are defined at the PSCB. Each PSCB is also associated with a bit position p. All leaves that can be reached from the PSCB through the 0-branch have a `0` at position p in the pattern, and the leaves that can be reached through the 1-branch have a `1` at position p. Furthermore, all leaves that can be reached from a PSCB will always have patterns at which bits 0 . . . p-1 are identical, i.e., the patterns start to differ at position p. The bit position associated with a PSCB is stored in the previous PSCB or in a DT entry and is called the NBT(i.e.,

next bit to test). The format of a PSCB entry is the same as the format of a DT entry. It is implemented in random access memory.

Detailed Description Text (37):
Thus, PSCBs are only inserted in the tree at positions where leaf patterns differ. This allows efficient search operations since the number of PSCBs, and thus the search performance, depends only on the number of leaves in a tree and not on the length of the patterns. The PSCB register format is depicted in FIG. 12.

Detailed Description Text (38):
In summary, a PSCB entry can be empty, can point to a leaf, or can point to another PSCB, and also contain the next bit to test (NBT) for that PSCB. FM PSCBs always have a shape defined by a width of one and a height of one, as described further below.

Detailed Description Text (40):
In the actual implementation, the key is inserted in a special key register 102. It is then hashed 104, and the results are stored in a hashed key register 106. The hash function 104 is programmable, and one of the functions is the null hash function (i.e., no hash). The first n bits of the hashed key are used as an index to the DT table 108. One programmable feature is the insertion of a bit vector right after the bits used to index in the DT entry. This bit vector is called a "color" value (register 124), and the result of the hashed key and the inserted color value is stored inside the hashed key register 106.

Detailed Description Text (41):
The format of a leaf in a FM tree contains control information including a pattern. The pattern identifies the leaf as unique in the tree. A leaf also contains the data needed by the application that initiated the tree search. The data contained in a leaf is application dependent and its size or memory requirements are defined by the LUDefTable entry for the tree. FIG. 13 illustrates the fixed leaf format for FM trees.

Detailed Description Text (42):
The steps in processing the DT entry are as follows: The DT entry is read from memory. If the DT entry is a null entry, this means that there are no leaves in the tree that have the same first "n" bits as the hashed key, so the search fails. If the DT entry has a pointer to a leaf, then the leaf is read from memory using the pointer from the DT 108 as the address of the leaf. The leaf is stored in a register and is compared with the key. This step is called compare at the end. If there is a full match, the tree search succeeds. Otherwise, the tree search fails. If the DT entry has a pointer to a PSCB 110 and an NBT, the NBT is first stored in a specific register. Then the NBT number is used to find the bit in the key in location NBT. That bit (0 or 1) is used along with the pointer to the PSCB to extract the correct PSCB entry: the bit is appended at the end of the pointer and that gives the full address in memory of the PSCB. The PSCB is read and stored in a specific register; the hardware then processes the PSCB entry. At this point, the algorithm is starting to walk down the is tree.

Detailed Description Text (43):
The steps in processing the PSCB entry are as follows: If the PSCB entry is a null entry, this means that there are no leaves in the tree that have the same first NBT bits as the key, so the search fails. If the PSCB has a pointer to a leaf, then the leaf is read from memory using the pointer from the PSCB as the address of the leaf. The leaf is stored in a register and is compared with the key. This step is called compare at the end. If there is a full match, the tree search succeeds. Otherwise, the tree search fails. If the PSCB has a pointer to a PSCB and an NBT, the NBT is first stored to the specific register, and this becomes the current NBT. Then this NBT number is used to find the bit in the key in location NBT. That bit (0 or 1) is used along with the pointer to the PSCB to extract the correct next

PSCB entry. The bit is appended at the end of the pointer and gives the full address in memory of the PSCB. The PSCB is read and stored in the specific register. Then the hardware will repeat this processing of a PSCB entry.

Detailed Description Text (44):
During the tree walk, not all bits of the leaf are tested, but only those bits for which there is a PSCB (branch in the tree). Therefore, once a leaf has been found, the pattern of the leaf must be compared with the key, to make sure that all bits match. This is the reason for the compare-at-the-end operation of the algorithm. Success or failure of the search is marked by an OK/KO flag, along with a completion flag. When the completion flag is triggered, the program or hardware that uses this FM tree search engine can examine the OK/KO flag.

Detailed Description Text (46):
One capability of the hardware is an automatic insert (a hardware insert) of a key. As the search for the (hashed) key proceeds, when there is a mismatch (KO), the leaf can be automatically inserted at that point by using the hardware to create the PSCB on the fly. In this case, the concept of the full match tree can be used as a cache.

Detailed Description Text (49):
For performance reasons, it is inefficient to read a DT entry only to find that it contains a pointer to a leaf, after which the leaf itself must be read. This situation will occur very often for FM trees, which have many single leaf entries per DT entry. The concept of a direct leaf allows a trade-off between more memory usage and better performance.

Detailed Description Text (50):
A tree can have direct leaves enabled, which is specified in the lookup definition table (LUDefTable). The difference between trees with direct leaves enabled and disabled is illustrated in FIG. 7. When direct leaves are enabled and a DT entry contains a single leaf, this leaf 130 is stored directly in the DT entry itself. Otherwise, the DT entry will contain a pointer to the leaf.

Detailed Description Text (51):
Shaping is a feature of the tree search memory (TSM) and is used to specify how an object, like a leaf or PSCB, is stored in the TSM. The shape is defined by the parameters width and height. The height of an object denotes the number of consecutive address locations at which the object is stored. The width of an object denotes the number of consecutive banks at which the object is stored. For width and height, the hardware automatically reads the appropriate number of locations. From a picocode point of view, an object is an atomic unit of access. The width must always be 1 for objects stored in SRAM. The width may be greater than 1 for objects in DRAM. Objects that are small enough to fit within a single memory location are defined to have a height of one and a width of one. The shape of a DT entry with direct leaves disabled is always (W=1, H=1). When the DT entry is stored in dynamic random access memory (DRAM), it occupies exactly 64-bits. The shape of a DT entry with direct leaves enabled equals the shape of the leaf, which is specified in the LUDefTable. In general, this causes more memory to be used by the DT 108. It also causes an impact of the leaf shape on the DT entry address calculation.

Detailed Description Text (52):
After a DT entry has been read and assuming the DT entry does not contain a direct leaf nor is it empty, the search continues by walking the tree that starts at the DT entry. The tree-walk may pass several PSCBs (pattern search control blocks), until a leaf has been reached.

Detailed Description Text (53):
When a PSCB is encountered during a search in an FM tree, the tree search engine

hardware 70 will continue the tree-walk on the 0-branch or the 1-branch, depending on the value of bit p of the HashedKey.

Detailed Description Text (54):
During a tree walk, not all bits of the HashedKey are tested, but only those bits for which there is a PSCB. Therefore, when a leaf has been found, the pattern in the leaf must still be compared with the HashedKey to make sure that all bits match. Note that it is the HashedKey that is stored in the leaf and not the original input key. When an FM leaf is found, the following operations are performed: Step 1: The leaf pattern is compared with the HashedKey. When a match occurs, the operation proceeds with Step 2. Otherwise, if the leaf contains a chain-pointer to another leaf, this leaf is read and the pattern is compared again with the HashedKey. Without a match and without an NLA field, the search ends with failure (KO). Step 2: If a vector mask is enabled, the bit with number VectorIndex is read from the leaf's vector mask. This bit is returned as part of the search result. The search ends with success (OK).

Detailed Description Text (55):
FIG. 10 illustrates the processing logic of the Full Match search algorithm of the present invention. Processing starts in logic block 1000 with reading of an input key. The input key is then run through a hash function as indicated in logic block 1002. Hashing at the input key into a hashed key is an option. The hash function is chosen such that the entropy is highest at the leftmost bits of the hashed key, i.e., those bits that are used to address a direct table. The hash function is reversible, i.e., there exists a reverse hash function that can transform the hashed key into the input key. Next, in logic block 1004, the direct table is read. The upper N bits (whereby N is configurable) of the hashed key are used as an index into the direct table. When the entry that has been read is empty, the search returns KO (no match found). This is indicated by termination block 1006. As indicated in decision block 1008, a determination is made as to whether or not the entry points to a leaf. If the DT entry points to a leaf, then as indicated in logic block 1010 the leaf is read. Otherwise, the DT entry points to a PSCB. In this case, the appropriate part of a PSCB is read as indicated in logic block 1012. For a full match search, a PSCB includes two entries: a 0-part and a 1-part. The previous PSCB (or DT entry) contains a bit number (NBT: next bit to test). The NBT selects a bit in the hashed key (i.e., 0 or 1) which selects which PSCB entry to use. The PSCB entry either contains a pointer to a leaf, or a pointer to another PSCB. Processing then loops back to decision block 1008. Once a leaf is found in decision block 1008, and read in logic block 1010, the pattern stored in the leaf is compared bit-wise with the hashed key as indicated by logic block 1014. If all bits match, as indicated in decision block 1016, the search returns OK (successful match) as indicated in termination block 1018. The contents of the leaf is then passed to the application. Otherwise, the search returns KO (failure) as indicated in termination block 1020. As an extension to this processing logic, a PSCB may consists of 2.sup.b entries, such that b bits from the hashed key select which entry to read from the PSCB. This increases performance at a cost of more memory usage. Processing then loops back to decision block 1008. Once a leaf is found in decision block 1008, and read in logic block 1010, the pattern stored in the leaf is compared bit-wise with the hashed key as indicated by logic block 1014. If all bits match, as indicated in decision block 1016, the search returns OK (successful match) as indicated in termination block 1018. The contents of the leaf is then passed to the application. Otherwise, the search returns KO (failure) as indicated in termination block 1020. As an extension to this processing logic, a PSCB may consists of 2.sup.b entries, such that b bits from the hashed key select which entry to read from the PSCB. This increases performance at a cost of more memory usage.

Detailed Description Text (56):
A cache can be used for increasing the search performance in trees. Use of a cache can be enabled in the LUDefTable on a per tree basis. During a search, the tree

search engine 70 will first check in the cache to determine if a leaf is present that matches the HashedKey. If such a leaf is found, it is returned and no further search is required. If such a leaf is not found, a normal search starts.

Detailed Description Text (57):
For the tree search engine hardware 70, a cache look-up is exactly identical with a normal search. Thus, the input key is hashed into a HashedKey, and a direct table 108 access is performed. The direct table 108 acts as a cache. When the cache search returns OK (success), the search ends. Otherwise, the tree search engine 70 starts a second search in the full tree--except that no hash operation is performed. The contents of the HashedKey register 106 are reused.

Detailed Description Text (59):
The tree search engine 70 provides hardware search operations in FM trees, LPM trees and SMT trees. For all tree types varying amounts of software are required to initialize and maintain a tree. Only FM trees and LPM trees have the capability that insertion and removal of leaves can be done without the intervention of control point processor 34. The use of this feature allows for a scalable configuration and still has the flexibility to allow the CP 34 to insert or remove leaves if needed.

Detailed Description Text (61):
FM trees are the best performing trees since they benefit significantly from the hashing function. The tree search engine provides multiple fixed hashing functions that offer very low collision rates. Assuming that the DT 108 is large enough, the probability of having multiple leaves associated with a single DT entry is very low. This is the 1+ epsilon rule, whereby epsilon represents the number of collisions in a DT entry. A DT entry with one leaf has an epsilon=0. Thus, with the hashing functions and using FM trees, the value of epsilon should be very small.

Detailed Description Text (62):
The structure of a DT entry in an FM tree can be seen in FIG. 8. Each DT entry is 36-bits wide and contains one of the following formats: Empty DT entry. There are no leaves associated with this DT entry. Pointer to next PSCB. The DT entry contains a pointer to a PSCB. The next PSCB address (NPA) and next bit to test (NBT) fields are valid. Pointer to leaf. There is a single leaf associated with the DT entry. The leaf control block address (LCBA) contains the pointer to this leaf. Direct leaf. There is a single leaf associated with a DT entry and the leaf is stored in the DT entry itself. The first field of a leaf must be the NLA rope, which implies that direct leaves must have the rope enabled. A rope is a circular linked list that is used to link leaves in a tree together. Picocode can "walk the rope" or sequentially inspect all leaves in a rope. It should be noted that the first two bits in the NLA are reserved to denote `10` such that they automatically encode "direct". direct leaves will only be used for a given tree if this is enabled in the LUDefTable. FM PSCBs have the same structure as an FM DT entry except that they consists of two PSCB lines, whereby each PSCB line can have one of the two formats shown in FIG. 8. The two PSCB lines are allocated consecutively in memory and are used as a branch for walking the tree. The next bit to test (NBT) field signifies the offset into the key to use as the bit comparison for walking the PSCBs and denotes which of the two PSCB lines to use.

Detailed Description Text (63):
An example of searching a FM tree can be seen in FIG. 9 where a 7-bit value is stored in the tree. The example is simplified by using the three most significant bits (MSB) of the key as a hash into the FM DT 108. There are five leaf entries (L0-L4)stored in this tree.

Detailed Description Text (64):
As a first example, assume a binary input key of 1110011. The first three bits `111` index into DT entry 7, where an LCBA (leaf control block address) pointing to

leaf L0 is present. The leaf L0 is read by the TSE 70 and the pattern in L0 is compared with the input pattern. In this example, an exact match occurs and the TSE will return OK (success).

Detailed Description Text (65):
Assume now an input pattern of 1001110. DT entry 4 contains a pointer to PSCB0 with an NBT field of 3. This means that the fourth bit in the key, `1` (bit 0 is the MSB or leftmost bit), determines which branch of the tree is taken. Since the fourth bit is a `1`, the bottom half of PSCB0 is used; had it been a `0`, the upper half of PSCB0 would have been used. Each PSCB is essentially a two element array of PSCB lines where an NBT value of `0` indexes into the first element and an NBT value of `1` indexes into the second element. Thus, the search continues because PSCB line 1 of PSCB0 contains an NBT of 6 and a next PSCB address (NPA) pointing to PSCB2. With an NBT of 7 and bit 7 of the input pattern equaling `0`, the upper half of PSCB2 is used containing a pointer to L3. Reading leaf L3 and performing the full compare operation of the pattern in L3 with the input pattern returns an OK (success).

Current US Original Classification (1):
707/6

Current US Cross Reference Classification (1):
707/10

Current US Cross Reference Classification (2):
707/3

Current US Cross Reference Classification (3):
707/5

Current US Cross Reference Classification (4):
707/7

CLAIMS:

1. A method for determining a full match for a variable length search key by a computer processing device, comprising the acts of: reading an input key as a search string; hashing the input key using a hash function to generate a hashed key; using the N most significant bits of the hashed key as an index into a table representing a plurality of root nodes of search trees wherein each non-empty entry contains a pointer to a next branch in the search tree or a leaf; determining if the pointer in a non-empty table entry points to a leaf or a next branch of the corresponding search tree; reading the next branch contents if the pointer does not point to the leaf of the corresponding search tree; reading the leaf contents when the leaf of a corresponding search tree is reached and comparing a pattern in the leaf with the hashed key to determine if the leaf pattern matches the hashed key; and returning the contents of the leaf found to the requesting application if the leaf pattern matches the hashed key.

2. The method for determining the full match of claim 1 wherein the table representing a plurality of root nodes of search trees contains 2.sup.N entries.

3. The method for determining the full match of claim 1 wherein the computer processing device is a network processor.

4. The method for determining the full match of claim 1 wherein the contents of the next branch of the corresponding search tree points to another next branch.

5. The method for determining the full match of claim 1 wherein the contents of the next branch points to the leaf of the corresponding search tree.

6. The method for determining the full match of claim 1 further comprising returning a no match found indication if the leaf pattern does not match the hashed key and does not contain a pointer to another leaf.

7. The method for determining the full match of claim 1 further comprising returning a no match found indication if the index into the table is to an empty entry.

8. The method for determining the full match of claim 1 further comprising appending the contents of a color register to the hashed key to provide a final hashed key.

9. The method for determining the full match of claim 1 further comprising appending a string of zeros to the hashed key to provide a final hashed key.

10. The method for determining the full match of claim 1 further comprising the act of terminating the search for the full match when the bit number of the next branch exceeds the length of the hashed key.

11. The method for determining the full match of claim 1 wherein the hash function used on the input key in a reversible hash function that can transform the hashed key into the input key.

12. The method for determining the full match of claim 1 further comprising the acts of: if the leaf contains a chain pointer to another leaf, reading a pattern stored in another leaf and comparing the pattern with the hashed key; returning an indication of no match found if the pattern stored does not match the hashed key and does not contain a pointer to a next leaf in the chain.

13. The method for determining the full match of claim 1 further comprising the acts of: if the leaf contains a chain pointer to another leaf, reading a pattern stored in another leaf and comparing the pattern with the hashed key; returning an indication of match found if the pattern stored does match the hashed key.

14. A computer readable medium containing a computer program product for determining a full match for a variable length search key, comprising: program instructions that read an input key as a search string; program instructions that hash the input key using a reversible hash function that can transform the hashed key into the input key; program instructions that use the N most significant bits of the hashed key as an index into a table representing a plurality of root nodes of search trees wherein each non-empty entry contains a pointer to a next branch in the search tree or a leaf; program instructions that determine if the pointer in a non-empty table entry points to a leaf or a next branch of the corresponding search tree; program instructions that read the next branch contents if the pointer does not point to the leaf of the corresponding search tree; program instructions that read the leaf contents when the leaf of a corresponding search tree is reached and compare a pattern in the leaf with the hashed key to determine if the leaf pattern matches the hashed key; and program instructions that return the contents of the leaf found to the requesting application if the leaf pattern matches the hashed key.

15. The computer program product for determining the full match of claim 14 wherein the table representing a plurality of root nodes of search trees contains 2.sup.N entries.

16. The computer program product for determining the full match of claim 14 wherein the computer processing device is a network processor.

17. The computer program product for determining the full match of claim 14 wherein the contents of the next branch of the corresponding search tree points to another

next branch.

18. The computer program product for determining the full match of claim 14 wherein the contents of the next branch points to the leaf of the corresponding search tree.

19. The computer program product for determining the full match of claim 14 further comprising program instructions that return a no match found indication if the leaf pattern does not match the hashed key and does not contain a pointer to another leaf.

20. The computer program product for determining the full match of claim 14 further comprising program instructions that return a no match found indication if the index into the table is to an empty entry.

21. The computer program product for determining the full match of claim 14 further comprising program instructions that append the contents of a color register to the hashed key to provide a final hashed key.

22. The computer program product for determining the full match of claim 14 further comprising program instructions that append a string of zeros to the hashed key to provide a final hashed key.

23. The computer program product for determining the full match of claim 14 further comprising program instructions that terminate the search for the full match when the bit number of the next branch exceeds the length of the hashed key.

24. The computer program product for determining the full match of claim 14 further comprising: program instructions that read a pattern stored in another leaf and compare the pattern with the hashed key if the leaf contains a chain pointer to another leaf; program instructions that return an indication of no match found if the pattern stored does not match the hashed key and does not contain a pointer to a next leaf in the chain.

25. The computer program product for determining the full match of claim 14 further comprising: program instructions that read a pattern stored in another leaf and compare the pattern with the hashed key if the leaf contains a chain pointer to another leaf; program instructions that return an indication of match found if the pattern stored does match the hashed key.

26. A method for determining a match for a variable length search key by a computer processing device, comprising the acts of: (a) reading an input key; (b) hashing the input key using a hash function to generate a hashed key; (c) using a portion of the hashed key as an address to access a table representing a plurality of nodes of search trees wherein at least one node includes a pointer to a next branch in the search tree; (d) reading the pointer; (e) accessing the next branch identified by said pointer; (f) repeating acts (d) and (e) until a leaf in said tree is reached.

27. The method of claim 26 further including the acts of reading a pattern stored at said leaf; correlating the pattern with the hashed key to determine if stored pattern matches the hashed key; and returning to a requester information stored in the leaf if the patterns match.

28. The method of claim 26 wherein the pointer includes a Next Bit To Test (NBT) field carrying an indicator identifying the bit to test in the hashed key and the Next Pointer Address (NPA) field carrying the address for the next pointer.

29. The method of claim 26 wherein the pointer further includes a control field carrying a code indicating a leaf is reached.

32. A method for determining a match for a variable length search key by a computer processing device, comprising the acts of: (a) reading an input key; (b) hashing the input key using a hash function to generate a hashed key; (c) using a portion of the hashed key as an address to access a table with entries representing a plurality of nodes of search trees and at least one entry including a pointer to a leaf.

33. The method of claim 32 further including the acts of reading a pattern stored at said leaf; correlating the pattern with the hashed key to determine if the stored pattern matches the hashed key; and returning to a requester information stored in the leaf if the patterns match.

34. A data structure for routing packets in a communications network comprising: a direct table having a plurality of entries wherein at least one entry represents a root node of a tree; at least a branch of said tree operably coupled to the root node; at least two pointers, one operably positioned in said root node, and the other operably positioned in the branch, said pointer including a Next Bit to Test (NBT) Field identifying a bit to be tested next in a key, Next Pointer Address (NPA) field carrying an address of a next pointer and a Control field; and a leaf operably coupled to said branch wherein said branch carries at least a pattern representative of a key to be tested.

35. A data structure comprising: a direct table having a plurality of entries wherein at least one entry represents a root node of a tree; a first pointer operably positioned at said root node, said first pointer including a control field, a Net Pointer Address Field (NPA) for storing a next address and a Next Bit To Test (NBT) field to carry information identifying a next bit in a data string to be tested; at least one branch of said tree operably coupled to the root node; a second pointer operably positioned at the branch, said second pointer including a two-element array with each array including control field, NPA field and NBT field; and at least one leaf operable coupled to the at least one branch.

Previous Doc Next Doc Go to Doc#

Freeform Search

Database: US Pre-Grant Publication Full-Text Database
US Patents Full-Text Database
US OCR Full-Text Database
EPO Abstracts Database
JPO Abstracts Database
Derwent World Patents Index
IBM Technical Disclosure Bulletins

Term: l17 and (full near match) and leaf

Display: 50 **Documents in Display Format:** - **Starting with Number** 1

Generate: ○ **Hit List** ◉ **Hit Count** ○ **Side by Side** ○ **Image**

Search | Clear | Interrupt

Search History

DATE: Sunday, September 26, 2004 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=OR			
L19	l17 and (full near match) and leaf	1	L19
L18	l17 and (full near match)	1	L18
L17	L14 and (fixed near size)	5	L17
L16	L14 and (fix$ near size)	5	L16
L15	L14 and vector$	1	L15
L14	L13 and comparison	7	L14
L13	L12 and (match$ near key)	7	L13
L12	L11 and key	17	L12
L11	L8 and (tree near structure)	21	L11
L10	L8 and l3	0	L10
L9	L8 and (leaf near data)	1	L9
L8	L6 and (external near memory)	26	L8
L7	L6 and (externanl near memory)	0	L7
L6	L5 and (tree near search$)	495	L6
L5	707/$.ccls.	13830	L5
L4	L3 and leaf	2	L4

L3	optimiz$ near (tree near structur$)	10	L3
L2	optimiz@ near (tree near structur$)	4	L2
L1	optimiz@ near (terr near structur$)	0	L1

END OF SEARCH HISTORY